Divide & Conquer Design Technique

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY

D & C Design Technique

The Divide & Conquer strategy can be described in general terms as follows:

- A problem input (instance) is divided according to some criteria into a set of smaller inputs to the same problem.
- The problem is then solved for each of these smaller inputs, either recursively by further division into smaller inputs or by invoking an ad hoc or priori solution. Ad hoc solutions are often invoked when the input size is smaller than some preassigned threshold value, i.e., sorting single element lists.
- Finally, the solution for the original input is obtained by expressing it in some form as a combination of the solutions for these smaller inputs.

D & C Design Technique

For convenience, we formulate Divide & Conquer (DANDC) as a procedure.

```
Procedure DANDC (p,q)
    Global n, A(1:n);                    // 1 ≤ p ≤ q ≤ n  //
    Integer m,p,q;
    If Small (p,q) then
        Return (G(p,q))
    Else m ← Divide (p,q)                // p ≤ m < q //
        Return(Combine(DANDC(p,m),DANDC(m+1,q)))
    Endif
End DANDC
```

D & C Design Technique

$$T(n) = 2T(n/2) + d(n)$$

A general case:

$$T(n) \in \begin{cases} g(n) & \text{if n is small (n < thershold)} \\ aT(n/b) + d(n) + c(n) & \text{Otherwise} \end{cases}$$

D & C Design Technique

Example: (Binary Search)

- It is desirable to search a sorted array for a *key*.
- We first check the middle element; if it is equal to *key*, then we are done.
- If it is less than *key*, then we perform a binary search on the upper half of the array
- and vice versa if it is greater than the *key*.

Recurrence relation for Binary search is

$$T(n) = T(n/2) + c$$

$T(n) = 1 + c.\lg n$ by using *the subtitution method*,

then $T(n)$ is $\Theta(\lg n)$.

D & C Design Technique

Example: (Binary Search)

```
function binsearch (A[1:n],key)
    if n =0 or key > A[n]
          then return n +1
    else return (binrec (A[1:n],key)

function binrec(A[i:j],x)
// Binary search for x in subarray A[i:j] with the promise that
    A[i-1] < x ≤  A[j] //
if i= j   then return i
    k ← (i+j) /2
if x ≤ A[k] then return binrec(A[i:k],x)
          else return binrec (A[k+1:j],x)
```

D & C Design Technique

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
-5	-2	0	3	8	8	9	12	12	26	31

x = 12 ≤ A[k]?

i	k			j	no
	i	k		j	yes
	i	j			yes
	ik	j			no
		ij	i =j: stop and return 8.		

D & C Design Technique

Example (Interpolation search): One improvement suggested for binary search is to try to guess more precisely where the key being sought falls within the current interval of interest (rather than blindly using the middle element at each step).

- This mimics the way one looks up a number in the telephone directory. This method is called interpolation search, which requires only a simple step modification to the program above.

  k = (i+j)/2 is derived from i/2- i/2 + i/2 + j/2 = i + ½(j-i).

- Interpolation search simply amounts to replacing ½ in this formula by an estimate of where the key might be based on the values available: ½ would be appropriate if *v (key)* were in the middle of the interval between a[i] and a[j], but we might have better luck trying

- k = i + [(v – a[i])/(a[j] – a[i])] * (j-i).

D & C Design Technique

Example (Interpolation search):

$$k = i + [(v - a[i])/(a[j] - a[i])] * (j-i)$$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	A	A	C	E	E	E	G	H	I	L	M	N	P	R	S	X

P ***R*** S X

S X

k= 1 + [(19-1)/(24-1)]*(17-1)=13, where i=1, j=17, a[i]=1, a[j]=24, and v=19 (index of S).

k = 14 + [(19 – 16)/(24-16)]*(17-14) = 15, where i = 14, j =17, a[i] = 16, a[j] = 24.

k = 16 + [(19 – 19)/(24-19)]*(17-16) = 16, where i = 14, j =17, a[i] = 16, a[j] = 24,

D & C Design Technique

Example (Interpolation search):

$$k = i + [(v - a[i])/(a[j] - a[i])] * (j-i)$$

1	2	3	4	5	6	7	8	9	10	11
-5	-2	0	3	8	8	9	12	15	26	31

i j $v = 12 \leq A[k]$?

k $k = 1 + [(12- (-5))/(31- (-5))]*(11-1) = 6$
$k = 7 + [(12- 9)/(31- 9)]*(11- 7) = 8$

ij i =j: stop and return 8.

D & C Design Technique

Example (Interpolation search):

- Interpolation search manages to decrease the number of elements examined to about lglgN.
- This is a very slowly growing function, which can be thought constant for practical purposes.
- If N is one billion, lglgN < 5.
- However, interpolation search requires some computation and assumes that the keys are rather well distributed over the interval.
- For small N, the lgN cost of straight binary search is close enough to lglgN that the cost of interpolating is not likely to be worthwhile.
- It certainly should be considered for large files, for applications where comparisons are particularly expensive, or for external methods where high access costs are involved.

D & C Design Technique

Example: (Max-Min problem)

- A Divide&Conquer approach is to break the array in half, find the max and min of each half, and then the combination step involves setting max to the larger of the two sub-maxes and min to the smaller of the two sub-mins.

- Recurrence Relation for Max-Min of size n is

  $T(n) = 2T(n/2) + 2; T(2) = 1.$

- If we solve this recurrence relation we find that $T(n) = 3n/2–2$, which means $T(n)$ is $\Theta(n)$.

D & C Design Technique

Example: (Max-Min problem)

```
function max-min (A[1:n],Lo,Hi,max,min)
     integer min1,min2,max1,max2;
if Hi= Lo
  then max = Hi, min = Lo
if Lo = Hi –1
     if A[Lo] < A[Hi]  then max = Hi, min = Lo
          else     max = Lo, min  = Hi
     else return (max-min(A[1:n],Lo,(Lo+Hi)/2,max1,min1)
          return (max-min(A[1:n],(Lo+Hi)/2 +1,Hi,max2,min2)
     return combine(A[1:n], min1,min2,max1,max2)

function combine(A[1:n], min1,min2,max1,max2)
if min1 < min2   then min = min1
else min = min2
if max1 > max2 then max = max1
else max = max2
```

D & C Design Technique

Example: (Merge Sort): The merge sort closely follows the D&C paradigm. Intuitively, it operates as follows:

1	2	3	4	5	6	7	8	9	10
380	285	179	652	351	423	861	254	450	520
|380	285|	179|	|652	351|	|423	861|	254|	450	520|
|285	380|		|351	652|	|423	861|		|450	520|
|179	285	380|	|351	652|	|254	423	861|	|450	520|
|179	285	351	380	652|	|254	423	450	520	861|
179	254	285	351	380	423	450	520	652	861

Analysis of MERGE-SORT:

Divide: computes the middle of the subarray, which takes constant time. Thus, $\Theta(1)$.

Conquer: we recursively solve two subproblems, each of size n/2, which contributes $2T(n/2)$ to the running time.

Combine: Merge procedure takes time $\Theta(n)$.

Then, if $n = 1$, $T(n) = \Theta(1)$,

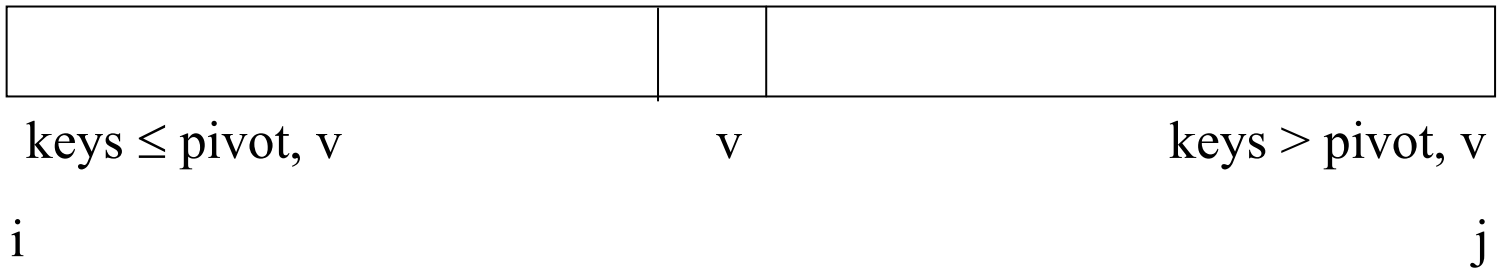
if $n > 1$, $T(n) = 2T(n/2) + \Theta(n)$.

$T(n)$ is $\Theta(n \lg n)$.

D & C Design Technique

Example: (Quick Sort): Quick sort is well known D&C algorithm for sorting. Here is the three-step D&C process for sorting a typical subarray A[p..r].

- *Divide*: The array A[p..r] is partitioned (rearranged) into two nonempty subarrays A[p..q] and A[q+1..r] such that each element of A[p..q] is less than or equal to each element of A[q+1..r]. The index q is computed as part of this partitioning process.
- *Conquer*: The two subarrays A[p..q] and A[q+1..r] are sorted by recursive calls to quıcksort.
- *Combine*: Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p..r] is now sorted.



D & C Design Technique

Example: (Quick Sort):

- A(1:n) is to be sorted. We permute the elements in the array so that for some i, all the records with keys less than v appear in A[1], …, A[i], and all those with key equal to v or greater appear in A[i+1], …, A[n] to sort both these groups of elements.

The following procedure implements the quick sort.

```
QUICKSORT (A,p,r)
if p< r
    then q ← PARTITION(A,p,r)
        QUICKSORT (A,p,q)
        QUICKSORT (A,q+1,r)
```

D & C Design Technique

- To sort an entire array A, the initial call is QUICKSORT(A,1,length[A]).
- The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p..r] in place.

```
PARTITION (A,p,r)
v ← A[p]
i ← p
j ← r
while TRUE
      do repeat j ← j –1
           until A[j] ≤ v
         repeat i ← i +1
           until A[i] ≥ v
         if i < j
           then exchange A[i] ↔ A[j]
         else break
repeat
      A[p] = A[j] ; A[j] = v
End
```

D & C Design Technique

1	2	3	4	5	6	7	8	9	10
$65_{=v}$	$70_{=j}$	75	80	85	60	55	50	$45_{=i}$	$+\infty$
65	45	7550	80 55	8560$_{=i}$	6085$_{=j}$	55 80	50 75	45 70	$+\infty$
65 60	45	7550	80 55	85**65**	6085$_{=j}$	55 80	50 75	45 70	$+\infty$
$60_{=v}$	$45_{=j}$	50	$55_{=i}+\infty$	**65**	$85_{=v}$	$80_{=j}$	75	$70_{=i}$	$+\infty$
$60_{=v}$	45	50	$55_{=i}$ $+\infty_{=j}$	**65**	$85_{=v}$	80	75	$70_{=i}$	$+\infty_{=j}$
$55_{=v}$	$45_{=j}$	$50_{=i}+\infty$	**60**	**65**	$70_{=v}$	$80_{=j}$	$75_{=i}+\infty$	**85**	$+\infty$
$50_{=v}$	$45_{=j}$	**55**	**60**	**65**	**70**	$80_{=v}$	$75_{=i}+\infty$	**85**	$+\infty$
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	$+\infty$

D & C Design Technique

Analysis of Quicksort: It is not straightforward to analyze the Quicksort since the division (partitioning) portion of the algorithm is dynamically dependent on the data being sorted and depends on the partitioning algorithm being used. The partition algorithm requires n comparisons, isolates a single pivot element.

- In the *worst* case, the pivot is at the end of the list and Quicksort is the same as Selection sort and its complexity is $\Theta(n^2)$. That is, the recurrence for the running time is

$$T(n) = T(n-1) + \Theta(n).$$

- To evaluate this recurrence, we observe that $T(1) = \Theta(1)$ and then iterate:

$$T(n) = T(n-1) + \Theta(n) = \sum_{1 \le k \le n} \Theta(k) = \Theta\left(\sum_{1 \le k \le n} k\right) = \Theta(n^2).$$

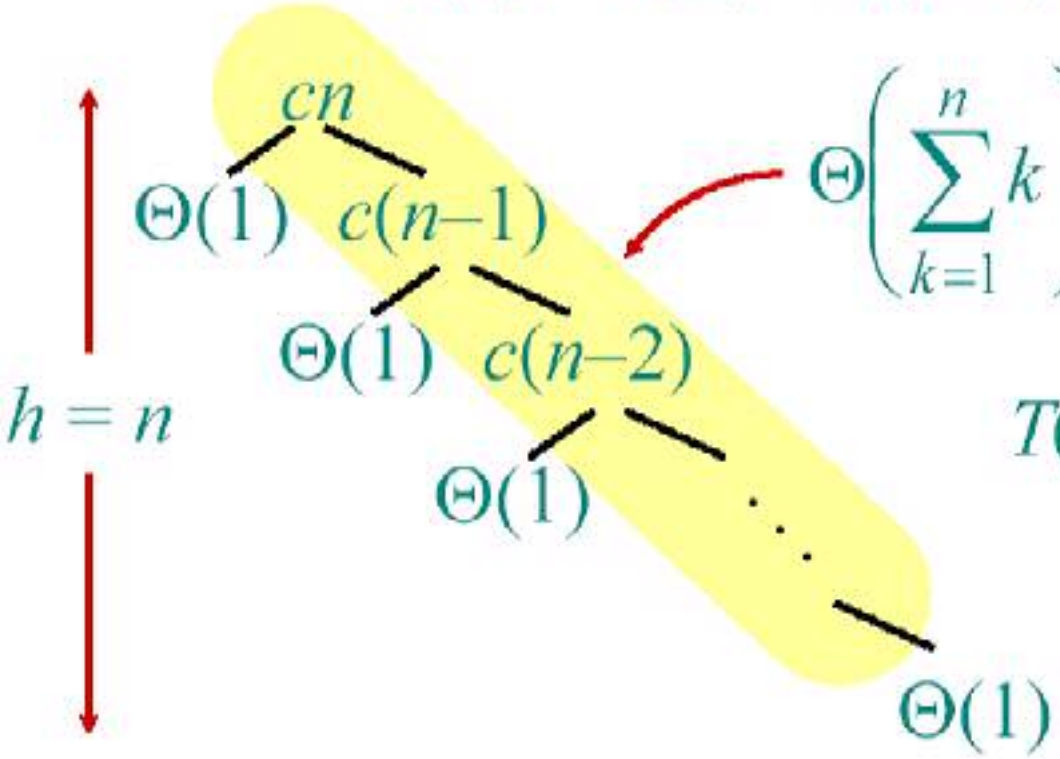
- At *best*, the pivot splits the list in half and the recurrence is

$$T(n) = 2.T(n/2) + c.n$$

which indicates a best behavior of $\Theta(n\lg n)$.

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

Best-case analysis

(For intuition only!)

If we're lucky, PARTITION splits the array evenly:

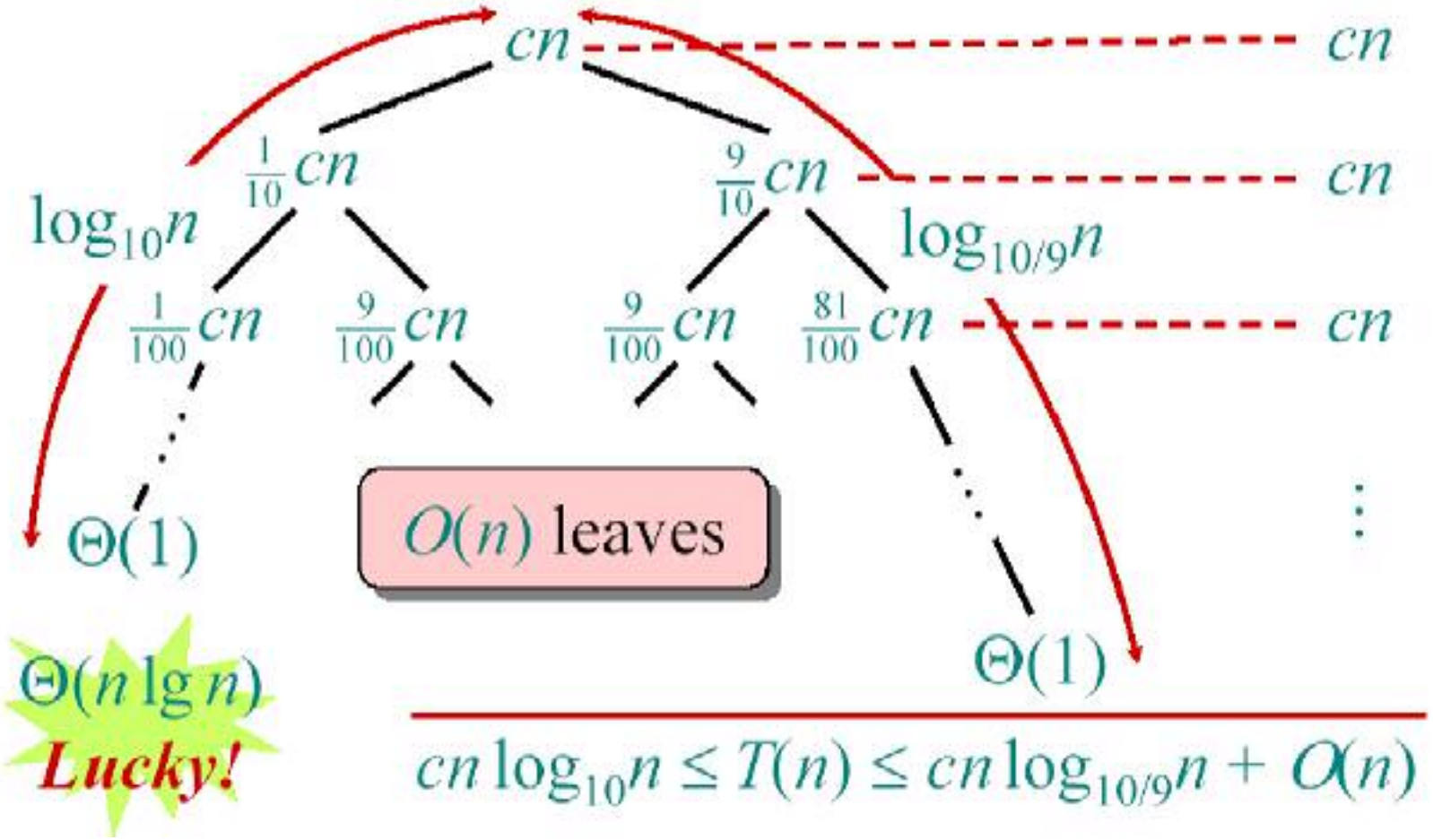
$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

Analysis of "almost-best" case



D & C Design Technique

Theorem: Quicksort takes a time in O(nlgn) to sort n elements on the average.

- For *average case* analysis, a statistical analysis can be performed on Quicksort over all n! possible permutations of the original data.
- The most natural assumption is that the elements of T are distinct and that each of the n! possible initial permutations of the elements is equally likely.

D & C Design Technique

Theorem: Quicksort takes a time in O(nlgn) to sort n elements on the average.

Proof:

- The assumption on the instance probability distribution:

  The pivot chosen by the algorithm when requested to sort T[1,…,n] lies with equal probability in any position with respect to the other elements of T.
- Each value has equal probability 1/n and the pivoting operation takes linear time, g(n) = n+1.
- It remains to sort recursively two sub-arrays of size i and (n-1-i).
- It can be shown that the probability distribution on the sub-arrays is still uniform.

D & C Design Technique

- Since the partitioning step takes about n+1 comparisons (the exact number depends on whether the "median of 3" or other improvements are applied), we can write the approximate running time function is as follows:

  $$T(n) = n+1 + (1/n) \sum_{0 \le i \le n-1} [T(i) + T(n-1-i)]$$

  Where i is the subscript of the element to the left of the pivot, $n \ge 2$, and $T(0) = T(1) = 0$. This translates into $T(2) = 3$.
- The above running time function is based on averaging over all possible values of i (note that both the best and worst cases are included in the average).
- When summing over i, each term appears twice. That is, for two different values of i, i and (n-1-i), are approximately the same.
- Therefore,

  $$T(n) = n+1 + (2/n) \sum_{0 \le i \le n-1} T(i), \text{ if } n \ge 2. \qquad (A)$$

D & C Design Technique

$T(n) = n+1 + (2/n) \sum_{0 \le i \le n-1} T(i)$, if $n \ge 2$. (A)

- Replacing n by n+1 gives:

  $T(n+1) = n+2 + (2/(n+1)) \sum_{0 \le i \le n} T(i)$, if $n \ge 1$. (B)

- Multiplying (A) by n results in:

  $nT(n) = n^2 + n + 2\sum_{0 \le i \le n-1} T(i)$ if $n \ge 2$. (C)

- Multiplying B by (n+1) gives

  $(n+1)T(n+1) = n^2 + 3n + 2 + 2\sum_{0 \le i \le n} T(i)$ if $n \ge 1$. (D)

- Computing (D) – (C), we obtain

  $(n+1)T(n+1) - nT(n) = 2n + 2 + 2T(n)$; $n \ge 2$.

  $(n+1)T(n+1) - (n+2)T(n) = 2(n + 1)$; $n \ge 2$. (E)

- Rearranging and simplifying (E) by dividing both sides by ((n+1)(n+2)

  $T(n+1)/(n+2) - T(n)/(n+1) = 2/(n + 2)$; $n \ge 2$.

D & C Design Technique

$T(n+1)/(n+2) - T(n)/(n+1) = 2/(n+2);\ n \geq 2.$

- Writing this recurrence for n = n-1, n-2, …, 3:

  $T(n)/(n+1) - T(n-1)/(n) = 2/(n+1)$

  $T(n-1)/(n) - T(n-2)/(n-1) = 2/(n)$

  $T(n-2)/(n-1) - T(n-3)/(n-2) = 2/(n-1)$

  …… …… ……..

  $T(4)/5 - T(3)/4 = 2/5$

  $T(3)/4 - T(2)/3 = 2/4$

- Summing these (n-1) equations gives:

  $T(n+1)/(n+2) - T(2)/3 = \sum_{4 \leq i \leq n+2} 2/i;\ n \geq 2,$

  Since T(2) = 3,

  $T(n+1) = (n+2)[1+ \sum_{4 \leq i \leq n+2} 2/i];\ n \geq 2,$

- Replacing (n+1) ($n \geq 2$) by n ($n \geq 3$), we get

  $T(n) = (n+1)[1+ \sum_{4 \leq i \leq n+1} 2/i];\ n \geq 3$

D & C Design Technique

$T(n)= (n+1)[1+ \sum_{4\le i\le n+1} 2/i]; n \ge 3$

- Now, we remove the $(n+1)^{th}$ term from the sum and subtract the 1^{st}, 2^{nd}, and 3^{rd} terms, which results in:

$T(n)= (n+1)+(n+1) \sum_{4\le i\le n+1} 2/i] =$

$(n+1)+(n+1)[\sum_{1\le i\le n} 2/i -1-2/3-1/2]$

$= (n+1)+(n+1)\sum_{1\le i\le n} 2/i - (n+1)-2(n+1)/3- (n+1)/2];$

$= (n+1)+(n+1)\sum_{1\le i\le n} 2/i - (n+1)-2n/3 - 2/3- n/2 - 1/2$

$= 2(n+1)\sum 1_{\le i\le n} 1/i -7n/6 - 7/6 = ;$

$T(n)= 2(n+1) \sum_{1\le i\le n} 1/i - 7n/6 - 7/6; n \ge 3,$

- Since $\sum_{1\le i\le n} 1/i$ is $\Theta(\lg n)$, Therefore, T(n) is $\Theta(n\lg n)$.
- We have shown that, on the average, Quicksort is an optimal sorting algorithm.
- In practice, it has been found to be the most efficient sorting method when a few modifications are included.

D & C Design Technique

Example (Selection of the k*th* Largest Element):

```
Procedure SelectKth (var A; k,first,last)
Begin
        PARTITION (A,first,last);
          If k = p, then pivot is the desired element
          Else if k < p, then SelectKth (A,k,first,first+p-2)
          Else SelectKth (A, k-p, first+p, last)
End;
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A_{=i}$	S	O	R	T	I	N	G	E	X	A	M	P	$L_{=j}$	$E_{=v}$
$A_{=i}$	S A	OE	R**E**	T	I	N	G	EO	X	A S	M	P	$L_{=j}$	$RE_{=v}$
				$T_{=i}$L	I	N	G	O	XP	SM	MS**R**	PX	TL	$SR_{=v}$
				$L_{=i}$	I	NG	GN**M**	O	$P_{=j}$	MN				
				L	I	G	**M**	O	P	N				

Matrix multiplication

Input: $A = [a_{ij}], B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$

$\Big\}\ i, j = 1, 2, \ldots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Standard algorithm

```
for i ← 1 to n
    do for j ← 1 to n
           do c_ij ← 0
              for k ← 1 to n
                  do c_ij ← c_ij + a_ik · b_kj
```

Running time $= \Theta(n^3)$

Divide-and-conquer algorithm

IDEA:

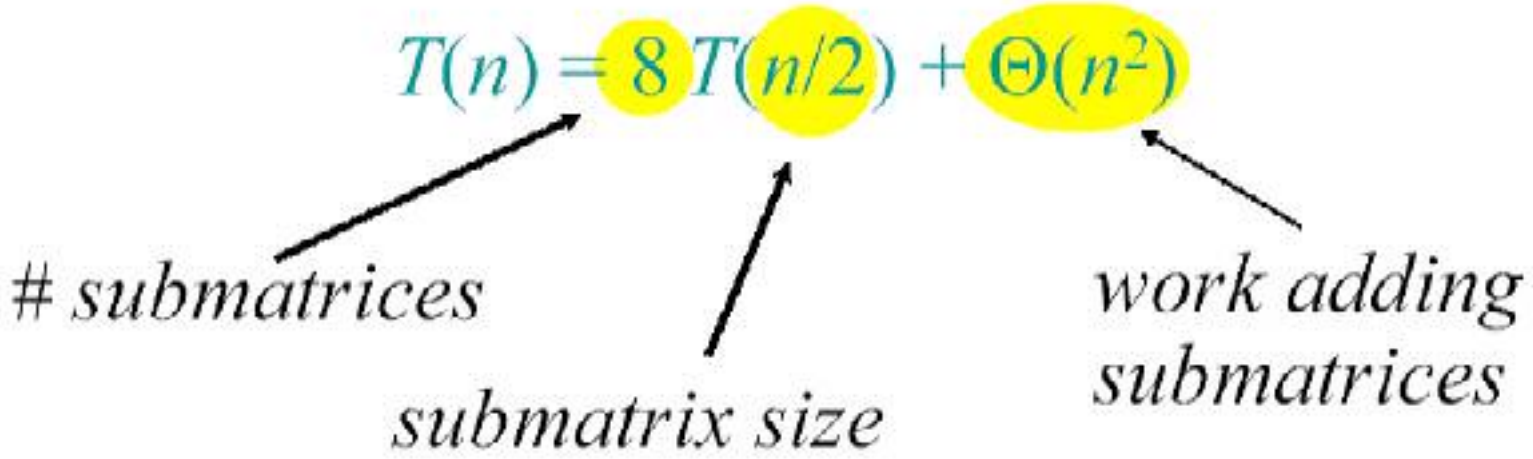
$n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dg \end{aligned}\right\} \begin{array}{l} 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$

Analysis of D&C algorithm



$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$
$$P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e$$
$$P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h)$$
$$P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

Strassen's algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Best to date (of theoretical interest only): $\Theta(n^{2.376\cdots})$.

D & C Design Technique

Example: (Multiprecision Multiplication of Integers)

• Multiplying n-figure large integers using the classic algorithm requires a time in $\Theta(n^2)$. By using D&C technique, we can reduce the multiplication of two n-figure numbers to four multiplications of n/2-figure numbers.

• More specifically, given x and y, which are both n-bit integers (assume that n is even), break x and y into two n/2-bit integers as follows:

$$x = a.2^{n/2} + b$$

$$y = c.2^{n/2} + d$$

Where a,b,c,and d are n/2-bits each. Then, the product of x and y can be represented in terms of products of a,b,c, and d as follows:

$$xy = ac2^n + (ad + bc).2^{n/2} + bd$$

• If we apply the same algorithm to the decimal numbers, then the multiplications will be:

$$xy = ac10^n + (ad + bc).10^{n/2} + bd$$

D & C Design Technique

Example: (Multiprecision Multiplication of Integers)
Let us multiply 981 * 1234 = 1210554.

• We split each operand into two halves:
0981 gives rise to a = 09 and b = 81, and 1234 to c = 12 and d = 34.

• Notice that $981 = 10^2a + b$ and $1234 = 10^2c + d$. Therefore, the required product can be computed as

$981 * 1234 = (10^2a + b) * (10^2c + d) = ac10^4 + (ad + bc)10^2 + bd =$
$09*12*10^4 + (09*34 + 81*12)*10^2 + 81*34$
$= 1080000 + 127800 + 2754 = 1210554.$

• As you see, the above procedure still needs four half-size multiplications: ac, ad, bc, bd.

• So we have reduced the n-bit product to four n/2-bit products and two shift and add operations. This process is applied recursively until only 1-bit products occur.

• We will show that the complexity of this algorithm is $\Theta(n^2)$.

D & C Design Technique

• We can apply an algebraic trick to reduce the problem to only three single precision multiplications as follows:

$$xy = ac2^n + [(a - b)(d - c) + ac + bd]\, 2^{n/2} + bd$$

The three subproblems are ac, (a-b)(d-c), and bd. Note that

$$[(a-b)(d - c) + ac + bd] = ad + bc$$

So the two formulas are equivalent.

Example:

ac = 09*12 = 108, bd = 81*34 = 2754, (a-b)*(d - c) = 90*46= 4140

Finally,

$$981 * 1234 = 108 * 10^4 + (4140 + 108 + 2734)10^2 + 2734$$
$$= 1080000 + 127800 + 2754 = 1210554$$

Thus, the product is reduced to three multiplications of two-figure numbers (09*12, 81*34, and 90*46) together with a certain number of shifts (multiplications by powers of 10), additions and subtractions.

D & C Design Technique

Recurrence Relations for MPI Multiplication

For the *four* subproblems case,

$T(n) = 4.T(n/2) + cn$; subject to $T(1) = 1$.

This assumes that the shift and add operations are of complexity proportional to n. If one solves this recurrence,

$T(n) = n^2 + c(n^2-n)$,

so $T(n)$ is $\Theta(n^2)$.

For the *three* subproblem case

$T(n) = 3.T(n/2) + cn$; subject to $T(1) = 1$.

If we solve this recurrence;

$T(n) = n^{1.59} + (1/2)cn(n^{0.59} - 1)$

$T(n) = n^{1.59} (1 + c/2) – cn/2$,

Therefore, $T(n)$ is $\Theta(n^{1.59})$, which is considerable lower growth rate than $\Theta(n^2)$.

D & C Design Technique

Conclusion

• Divide and conquer is just one of several powerful techniques for algorithm design.

• Divide-and-conquer algorithms can be analyzed using recurrences and the master method or characteristic equation method.

• D&C can lead to more efficient algorithms

• However, although the Divide & Conquer approach becomes more worthwhile as the instance to be solved gets larger, it may in fact be slower than the classic algorithm on instances that are too small.