DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Module 3
Radhika Chapaneri
Sem V

Control Abstraction of Divide and Conquer

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

Quick Sort

This algorithm is called *quicksort or Partition exchange sort*

1. Pick an element (**pivot**)
2. **Partition** the array into elements < pivot, = to pivot, and > pivot
3. Quicksort these smaller arrays separately

Quick Sort

• We are starting with an un-sorted arrayA[p…r]

• Divide: We start by dividing the array into two subarrays A[p…q-1] and A[q+1…r]. (for now, we just leave A[q]). We divide it such that all elements of A[p..q-1] are smaller than or equal to A[q], which again is smaller than or equal to A[q+1…r]. q is determined as a part of this step.

• Conquer: Sort the two arrays A[p..q-1] and A[q+1..r] by recursively calling quicksort.

• Combine: We are finished, since the array is now sorted.
The most difficult is probably ”Partition”, so here is an example:

QuickSort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

QuickSort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4

i =-1	P, j =0	1	2	3	4	5	6	r=7
	2	8	7	1	3	5	6	**x =4**

P j, i =0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4

i =0	J=1	2	3	4	5	6	7
2	**8**	7	1	3	5	6	**x= 4**

PARTITION(A, p, r)
```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

i =0	1	J=2	3	4	5	6	7
2	8	**7**	1	3	5	6	**x= 4**

i =0	1	2	J=3	4	5	6	7
2	8	7	**1**	3	5	6	**x= 4**

0	i=1	2	J=3	4	5	6	7
2	**1**	7	**8**	3	5	6	**x= 4**

0	i=1	2	3	J=4	5	6	7
2	1	7	8	**3**	5	6	**x= 4**

PARTITION(A, p, r)
```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	i=2	3	J=4	5	6	7
2	1	**3**	8	**7**	5	6	**x= 4**

0	1	i=2	3	4	J=5	6	7
2	1	3	8	7	**5**	6	**x= 4**

0	1	i=2	3	4	5	J=6	7
2	1	3	8	7	5	**6**	**x= 4**

0	1	i=2	3	4	5	J=6	7
2	1	3	**4**	7	5	6	**8**

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Return 3

QuickSort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Quick Sort Analysis

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.
- **Worst-case partitioning**
- The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n -1 elements and one with 0 elements.
- $T(n) = T(n-1) + n$
- $= \theta(n^2)$

Quick Sort Analysis

- **Best-case partitioning**
- In the most even possible split, PARTITION produces two subproblems,
- $T(n) = 2T(n/2) + n = \theta(n \log n)$

Divide-and-Conquer approach

- – For n≤ 2, make 1 comparison
- – For large n, divide set into two smaller sets and determine largest/smallest element for each set
- – Compare largest/smallest from two subsets to determine smallest/largest of combined sets
- –Do recursively

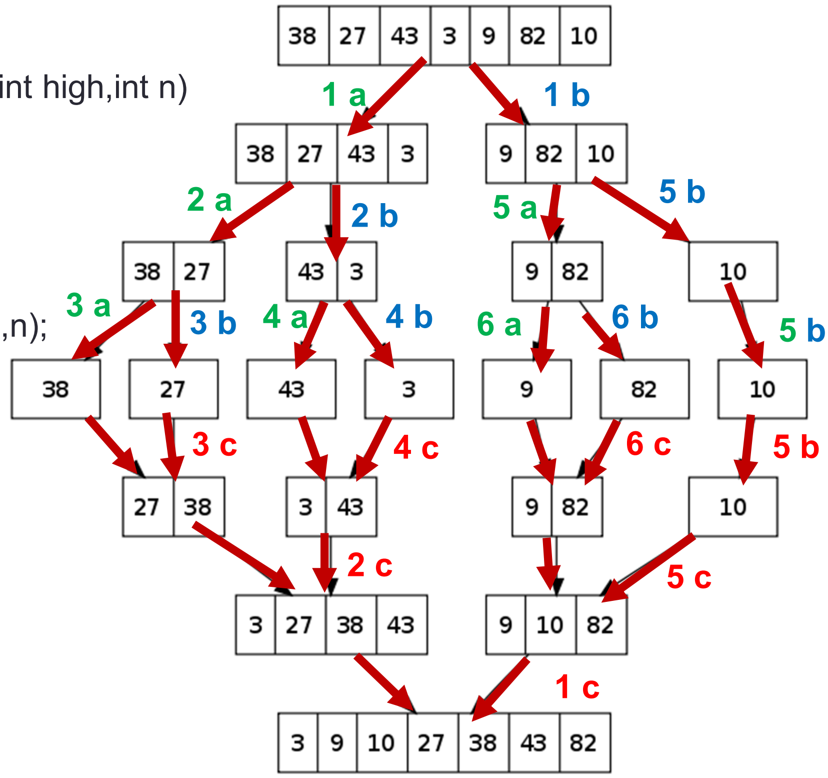
Mergesort

- void mergesort(int a[],int low,int high,int n)
- {
  - if(low<high)
  - {
-         int mid = (low +high)/2;
-         mergesort(a,low,mid,n);
-         mergesort(a,mid+1,high,n);
-         merge(a,low,mid,high);
-     }
- }

Mergesort

```
void mergesort(int a[],int low,int high,int n)

    if(low<high)
    {
     int mid = (low +high)/2;
  a  mergesort(a,low,mid,n);
  b  mergesort(a,mid+1,high,n);
  c  merge(a,low,mid,high);
 }
```



0	1	2	3	4	5	6
3	27	38	43	9	10	82

0 P	1	2	3 q
3	27	38	43

4 q+1	5	6 r
9	10	82

0	1	2	3	4	5	6k
3	9	10	27	38	43	82

- Merge(array A, int p, int q, int r)
- { array B[p..r] //temp array taken
- i = k = p // initialize pointers
- j = q+1
- while (i <= q and j <= r)
- { if (A[i] <= A[j])
- B[k++] = A[i++]
- else B[k++] = A[j++] }
- while (i <= q) B[k++] = A[i++] // copy any leftover to B
  while (j <= r) B[k++] = A[j++]
- for i = p to r A[i] = B[i] // copy B back to A }

Analysis of Merge Sort

- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Recurrence equation of Merge sort

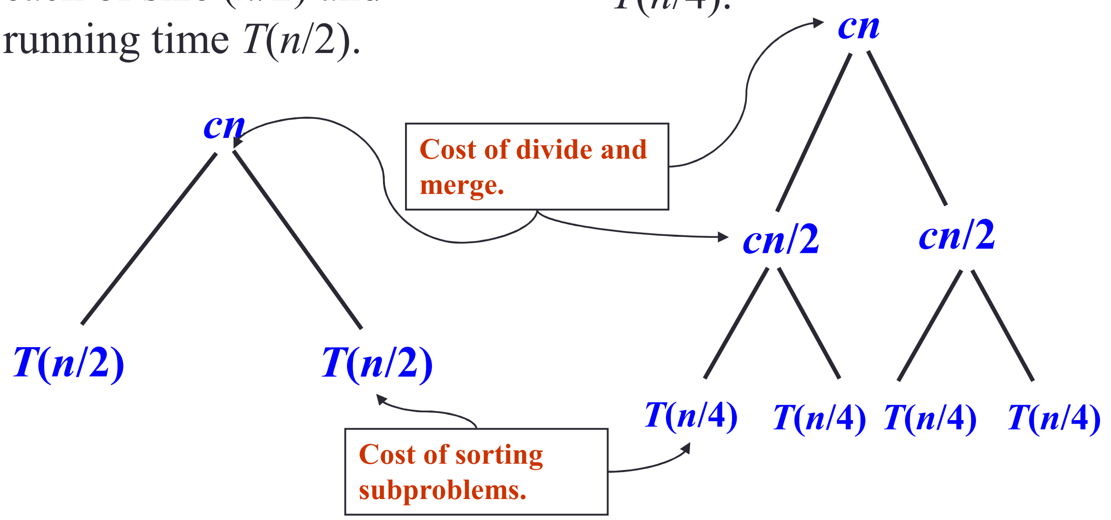
$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

- Complexity of merge sort = O(nlogn)

Recursion Tree for Merge Sort

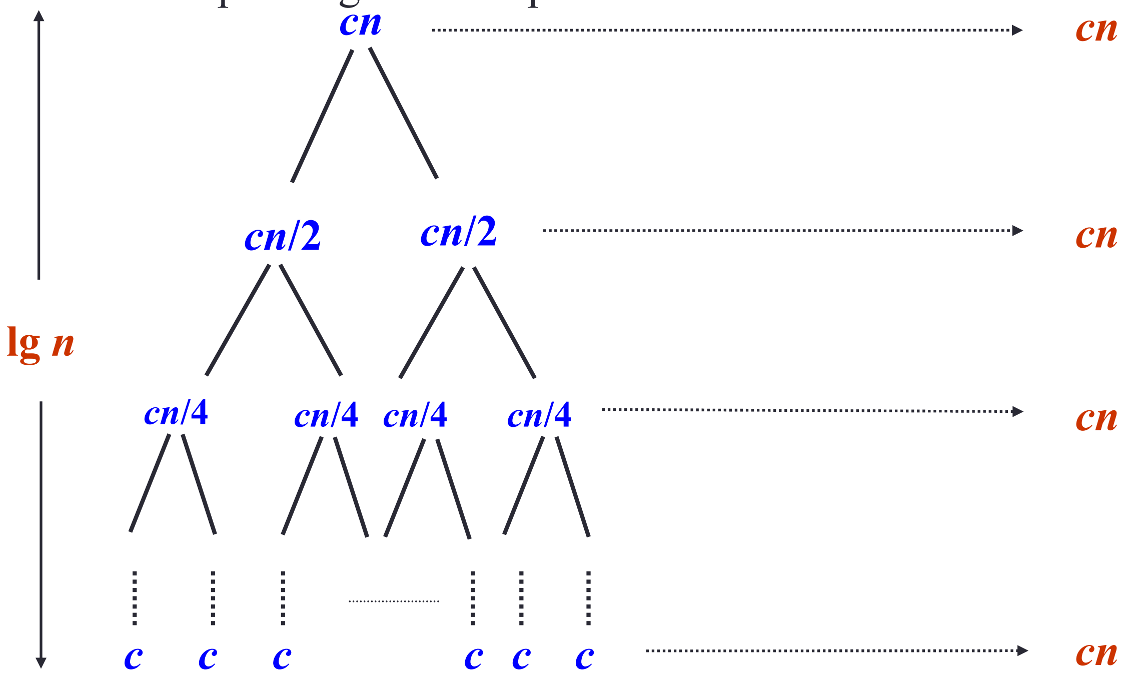
For the original problem, we have a cost of cn, plus two subproblems each of size $(n/2)$ and running time $T(n/2)$.

Each of the size $n/2$ problems has a cost of $cn/2$ plus two subproblems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

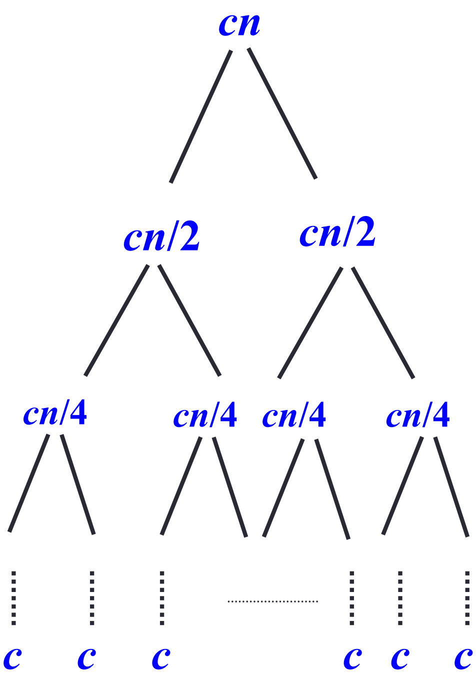
Continue expanding until the problem size reduces to 1.



Total : $cn\lg n + cn$

Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves $\Rightarrow$ *cost per level remains the same*.
- There are $\lg n + 1$ levels, height is $\lg n$. (Assuming n is a power of 2.)
  - Can be proved by induction.
- Total cost = sum of costs at each level = $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$.

Merge Sort

- Although merge sort has an optimal complexity, it needs an additional space of O(n) for the temporary array.

Matrix multiplication

- Let A, B and C be n × n matrices

C = AB

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

Matrix multiplication

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

- The straightforward method to perform a matrix multiplication requires $O(n^3)$ time.

Divide-and-conquer approach

- We use a divide-and-conquer algorithm to compute the matrix product C= A **.** B, we assume that n is an exact power of 2 in each of the n x n matrices.
- Suppose that we partition each of A, B, and C into four n/2 n/2 matrices
-

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Divide-and-conquer approach

- We rewrite the equation C = A . B as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

This equation corresponds to the four equations

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Divide-and-conquer approach

- The recurrence equation for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE is :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Strassen's matrix multiplicaiton

- The key to Strassen's method is to make the recursion tree slightly less bushy.
- That is, instead of performing eight recursive multiplications of n/2 X n/2 matrices, it performs only seven.
- Strassen's method involves first computing the seven n/2 x n/2 matrices P, Q, R, S, T, U, V

Strassen's matrix multiplicaiton

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$
  $Q = (A_{21} + A_{22})B_{11}$
  $R = A_{11}(B_{12} - B_{22})$
  $S = A_{22}(B_{21} - B_{11})$
  $T = (A_{11} + A_{12})B_{22}$
  $U = (A_{21} - A_{11})(B_{11} + B_{12})$
  $V = (A_{12} - A_{22})(B_{21} + B_{22})$.

- $C_{11} = P + S - T + V$
  $C_{12} = R + T$
  $C_{21} = Q + S$
  $C_{22} = P + R - Q + U$

Time complexity

- Time complexity:
- The recurrence for the running time T(n) of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Example

- Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$