Agenda:

- Divide and Conquer technique
  - The classic example: Merge Sort (CLRS p.30-39)
  - Exponentiation (CLRS p.956-958 [$\bmod p$])
  - Karatsuba's algorithm for multiplying large integers
  - Strassen's algorithm for matrix multiplication (CLRS Ch.4.2)

Divide and Conquer :

- To solve a problem:
  - Break problem into smaller subproblems (Divide)
  - Solve each subproblem recursively
  - Solve the problem for the entire instance using partial solutions (Conquer)

Divide and Conquer and recursive programs

- A useful design technique for algorithms is *divide-and-conquer*
- These algorithms are often recursive and consist of the following steps:
  - Divide: Partition the input into two or more disjoint (smaller) pieces & *recursively* solve the subproblems
  - Conquer: Leverage on the solutions for the subproblems to get a solution for the original problem.
  - (Of course, if input's size is small, just "conquer" using a simple method.)
- To analyze (the running time of) a recursive program we express their running time as a recurrence
- We then solve the recurrence to find a closed form for the running time of the algorithm.

Merge-Sort

`Merge`$(A; lo, mid, hi)$

  `**pre-condition:` $lo \le mid \le hi$

  `**pre-condition:` $A[lo, mid]$ `and` $A[mid+1, hi]$ `sorted`

  `**post-condition:` $A[lo, hi]$ `sorted`

`Merge-Sort`$(A; lo, hi)$

 `if` $(lo < hi)$ `then`

  $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$

  `Merge-Sort`$(A; lo, mid)$

  `Merge-Sort`$(A; mid+1, hi)$

  `Merge`$(A; lo, mid, hi)$

```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Merge sort, the big idea — divide-and-conquer:

- Divide the whole array into 2 subarrays of equal size;
- Recursively merge sort the 2 subarrays;
- `Merge`: Combine the 2 sorted subarrays into a sorted array
  - Copy $A[lo, ..., mid]$ (or $A[p, q]$) to array L
  - Copy $A[mid + 1, .., hi]$ (or $A[q + 1, r]$) to array R
  - If we have still haven't exhausted all elements in L and in R: Copy the smallest of $L[i]$, $R[j]$ into $A[k]$ and advance k and either i or j (depends on which element was copied)
  - Once we traversed all elements of either L or R – copy the remaining elements in the non-exhausted array
    (If we exhausted all of L, copy remaining elements from R; if we exhausted R, copy all remaining elements in L.)
  - CLRS version: avoids checking that either L or R have been exhausted using the clever trick of a sentinel ∞ (some dummy element greater than all elements in $L \cup R$)
- An example:

	1	2	3	4	5	6	7	8	9	10	11	12	13
A	[31	23	01	17	19	28	09	03	13	15	22	08	29]

Merge sort — Example:

$\boxed{31\ 23\ 01\ 17\ 19\ 28\ 09\ 03\ 13\ 15\ 22\ 08\ 29}$

$\boxed{31\ 23\ 01\ 17\ 19\ 28\ 09}$ $\boxed{03\ 13\ 15\ 22\ 08\ 29}$

$\boxed{31\ 23\ 01\ 17}$ $\boxed{19\ 28\ 09}$ $\boxed{03\ 13\ 15}$ $\boxed{22\ 08\ 29}$

$\boxed{31\ 23}$ $\boxed{01\ 17}$ $\boxed{19\ 28}$ $\boxed{09}$ $\boxed{03\ 13}$ $\boxed{15}$ $\boxed{22\ 08}$ $\boxed{29}$

$\boxed{31}$ $\boxed{23}$ $\boxed{01}$ $\boxed{17}$ $\boxed{19}$ $\boxed{28}$ $\boxed{03}$ $\boxed{13}$ $\boxed{22}$ $\boxed{08}$

$\boxed{23\ 31}$ $\boxed{01\ 17}$ $\boxed{19\ 28}$ $\boxed{03\ 13}$ $\boxed{08\ 22}$

$\boxed{01\ 17\ 23\ 31}$ $\boxed{09\ 19\ 28}$ $\boxed{03\ 13\ 15}$ $\boxed{08\ 22\ 29}$

$\boxed{01\ 09\ 17\ 19\ 23\ 28\ 31}$ $\boxed{03\ 08\ 13\ 15\ 22\ 29}$

$\boxed{01\ 03\ 08\ 09\ 13\ 15\ 17\ 19\ 22\ 23\ 28\ 29\ 31}$

Merge Sort Correctness

- Claim 1: `MergeSort` correctly sorts all arrays of size n.
- Proof: By induction.
- Base case: $n = 1$. Trivially, input is sorted and `MergeSort` does nothing.
- Induction Step: Use full induction.
  Fix $n > 1$. Assuming that for any array of size i, $1 \leq i < n$, `MergeSort` sorts it correctly, we show it also sorts correctly an array of size n.
  First note that since $n > 1$ then $lo < hi$. This means that:
  (1) $mid = \lfloor \frac{lo+hi}{2} \rfloor \leq \frac{lo+hi}{2} < \frac{hi+hi}{2} < hi$, so $A[lo, mid]$ has fewer elements than $A[lo, hi]$.
  (2) Similarly, $mid + 1 = \lfloor (lo + hi)/2 \rfloor + 1 \geq \lfloor (lo + lo)/2 \rfloor + 1 \geq lo + 1$ so $A[mid + 1, hi]$ has fewer elements than $A[lo, hi]$.
  Hence, IH implies that each of the recursive calls `Merge-Sort`$(A; lo, mid)$, `Merge-Sort`$(A; mid + 1, hi)$ sorts the respective part of the array. The following claim concludes the proof.
- Claim 2: Given an array A and 3 indices $lo \leq mid < hi$ such that $A[lo, mid]$ and $A[mid + 1, hi]$ are both sorted, `Merge`(A, lo, mid, hi) sorts all elements in $A[lo, hi]$.
- How to prove Claim 2?
  - 3 loops in the code, so use 3 LIs. (State and prove them formally!)
  - LI1 + LI2 : the invariants of copying one array onto another
  - LI3: $A[p, k - 1]$ contains the smallest $(k - p + 1)$ elements of $L \cup R$ in order, and $L[i, n_1] \cup R[j, n_2]$ contain the remaining $r - k$ $\big(= (r - p + 1) - (k - p + 1)\big)$ elements.

Recurrence relations — Merge Sort analysis

- MergeSort:
  - Divide the whole list into 2 sublists of equal size; recursively sort each sublist;
  - Merge the 2 sorted sublists into a sorted list.
- Let $T(n)$ denote $\#$KC for a list of size n
- Assumptions:
  - n (number of keys in the whole list) is a power of 2;
    This makes the analysis easier (since each time we are dividing by 2)
- Deriving recurrence relation:
  - Merge sort on 2 sublists $2 \times T(\frac{n}{2})$
  - Assembling needs $n-1$ KC (in the WC)
  - $T(n) = \begin{cases} 0 & , \quad \text{if } n = 1 \\ (n-1) + 2 \cdot T(\frac{n}{2}) & , \quad \text{otherwise} \end{cases}$
- How to solve this?
- Master Theorem (case 2): $T(n) = \Theta(n \log(n))$.

Divide and Conquer and More!

- It turns out that the idea of using multiple recursions on a partition of the instance is a very helpful idea.
  - It reduced the naïve sorting from $O(n^2)$ to $O(n\log(n))$.
  - We will later see a similar D&C idea with `QuickSort`. There are other problems when D&C give an immediate improvement over the naïve algorithm.
- But there are also case where the D&C idea is just the first step.
- The second step is to seek how to reduce the number of recursive calls.
- Remember our toy example:

```
procedure QZ(n)
if (n > 1) then
    a ← n × n + 37
    b ← a × QZ(n/2)
    return QZ(n/2) × QZ(n/2) + n
else
    return n × n
```

with runtime $T(n) = \begin{cases} 1, & \text{if n=1} \\ 5 + 3T(n/2), & \text{o/w} \end{cases}$ that solved to $\Theta(n^{\log_2(3)})$

Divide and Conquer — Reducing No. of Recursive Calls

- Remember our toy example, QZ(n) with runtime $\Theta(n^{\log_2(3)})$.
- Now consider the alternative:

```
procedure QZ(n)
if (n > 1) then
    a ← n × n + 37
    x ← QZ(n/2)
    b ← a × x
    return x × x + n
else
    return n × n
```

- The number of arithmetic operations now is:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 5 + T(\frac{n}{2}), & \text{o/w} \end{cases}$$

- Master theorem: $a = 1$, $b = 2$, $f(n) = 5 \in \Theta(n^0\,(\log(n))^0)$ so case 2 applies and we get $T(n) = \Theta(n^0\,(\log(n))^1) = \Theta(\log(n))$
- Note the dramatic improvement: from $n^{1.58}$ to $\log(n)$.

Example 1: Exponentiation

- Given integers b, n, want to compute $b^n \mod p$.
- This problem has application in cryptography (we compute power mod p, more details in CMPUT 304).
- Assume that n is a huge integer with hundreds of bits (e.g. 1024 bits).
- Naive approach: multiply b with itself n times (using a for-loop)
- We are doing n multiplication
  - If each multiplication take $O(1)$ time — overall $O(n)$ time.
- Fine, let's do a recursive divide-and-conquer call

```
procedure exp(b, n)
if (n = 0) then
   return 1
else
   return exp(b, ⌈n/2⌉) × exp(b, ⌊n/2⌋)
```

- The recurrence relation we get

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor), & \text{o/w} \end{cases}$$

or, if n is even: $T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1 + 2T(\frac{n}{2}), & \text{o/w} \end{cases}$

This solves to $T(n) = n$ (Master Theorem, case 1). (no improvement)

Example 1: Exponentiation

- Observation:
  For even n — reduce the number of recursive call by saving the result of `exp`$(b, \frac{n}{2})$, and squaring it.
  For odd n — `exp`$(b, \lceil\frac{n}{2}\rceil) = b \times$ `exp`$(b, \lfloor\frac{n}{2}\rfloor)$, so save `exp`$(b, \lfloor\frac{n}{2}\rfloor)$, square it, and make one more multiplication with b.
- Note that taking square of a number only takes one multiplication.
  I.e., we reduce the number of recursive calls by adding more less-costly operations (in this case, multiplications), and the runtime has vastly improved.
  E.g., to compute b^{50} we need only 7 multiplications (instead of 50 multiplications, naïvely): $b^{25} \cdot b^{25}$; $b \cdot b^{24}$; $b^{12} \cdot b^{12}$; $b^6 \cdot b^6$; $b^3 \cdot b^3$; $b \cdot b^2$; $b \cdot b$
- <u>procedure Power(b, n)</u>

```
if (n = 0) then
   return 1
else
   if (n is odd) then
      x ← Power(b, n − 1)
      return x × b        ** inductively, x = b^(n−1) so x · b = b^n
   else
      x ← Power(b, n/2)
      return x × x        ** inductively, x = b^(n/2) so x · x = b^(n/2 + n/2) = b^n
```

Example 1: Exponentiation

- ```
  procedure Power(b, n)
  if (n = 0) then
     return 1
  else
     if n is odd then
        x ← Power(b, n − 1)
        return x · b
     else
        x ← Power(b, n/2)
        return x · x
  ```
- Let $T(n)$ be the number of multiplications required to compute b^n.
- Assume $n = 2^k$ for some $k \geq 1$.

$$T(n) = T(\frac{n}{2}) + 1 = T(\frac{n}{4}) + 1 + 1 = \ldots = T(\frac{n}{2^k}) + k = k + 1$$

- Now assume $n = 2^k - 1$ for some $k \geq 1$.

$$\begin{aligned} T(n) &= T(n-1) + 1 = T(\frac{n-1}{2}) + 1 + 1 = T(2^{k-1} - 1) + 2 \\ &= T(2^{k-1} - 2) + 3 = T(2^{k-2} - 1) + 4 \\ \ldots &= T(1) + 2k = 2k + 1 \end{aligned}$$

- Therefore, $T(n) \in O(\log n)$.
  ```

Example 2: Multiplication of large integers :

- Suppose we are dealing with integers that have hundreds of bits (e.g. 256, 512, 1024 or 2048 bits).
- The naive algorithm for multiplication, the elementary algorithm takes $O(n^2)$ steps.
- Goal: do it faster, i.e. $o(n^2)$.
- Suppose that I and J are the two n bit integers to be multiplied.
- Break I into two parts: w denotes the $\frac{n}{2}$ MSBs, x denotes the $\frac{n}{2}$ LSBs.

$$I = \begin{array}{|c|c|}\hline \quad w \quad & \quad x \quad \\ \hline\end{array}$$

  So $I = w \cdot 2^{n/2} + x$.

- Similarly, we denote $J = y \cdot 2^{n/2} + z$.

$$J = \begin{array}{|c|c|}\hline \quad y \quad & \quad z \quad \\ \hline\end{array}$$

- It is easy to see that $I \cdot J = w \cdot y \cdot 2^n + (w \cdot z + x \cdot y)2^{n/2} + x \cdot z$.

Example 2: Multiplication of Large Integers (cont'd)

- $I \cdot J = w \cdot y \cdot 2^n + (w \cdot z + x \cdot y)2^{n/2} + x \cdot z$.
- In binary: Multiplying by $2^i \Leftrightarrow$ left-shift i bits; each left-shift takes $O(1)$ time.
- So to multiply by 2^n, and $2^{n/2}$ (for the second term), and add the results: $O(n)$ time.
- We have 4 multiplications of integers of $\frac{n}{2}$ bits each: $w \cdot y$, $w \cdot z$, $x \cdot y$, and $x \cdot z$.
- So, the time required for multiplying I and J is: $T(n) = 4T(\frac{n}{2}) + n$.
- Using master theorem: $T(n) \in \Theta(n^2)$.
- This is not better than the naive algorithm...

Example 2: Karatsuba's Algorithm for Multiplying Large Integers

- $I \cdot J = w \cdot y \cdot 2^n + (w \cdot z + x \cdot y)2^{n/2} + x \cdot z$.
- The bottleneck here is: too many recursive calls
  Let's aim to make ≤ 3 recursive calls to multiply two $\frac{n}{2}$-bit integers.
- **Observation:** Let $r = (w + x)(y + z) = w \cdot y + (w \cdot z + x \cdot y) + x \cdot z$.
- So r contains all 3 terms we need to compute $I \cdot J$, but not individually.
- So here's the plan:
  1. Compute $a \leftarrow w + x$. (Addition in time $O(n)$)
  2. Compute $b \leftarrow y + z$. (Addition in time $O(n)$)
  3. Recurse to find $c \leftarrow w \cdot y$. (recursive call on two $\frac{n}{2}$-bits integers)
  4. Recurse to find $d \leftarrow x \cdot z$. (recursive call on two $\frac{n}{2}$-bits integers)
  5. Recurse to find $r \leftarrow a \cdot b$. (recursive call on two $\frac{n}{2}$-bits integers)
  6. Compute $e \leftarrow r - c - d$. (Addition / subtraction in time $O(n)$)
  7. Do left-shifts and return $2^n \cdot c + 2^{n/2} \cdot e + d$. (Shift / addition in time $O(n)$)
- Recursive formula for this algorithm's run-time: $T(n) = 3T(\frac{n}{2}) + O(n)$
- Using Master theorem: $T(n) \in \Theta(n^{\log_2 3})$. Thus:
  Theorem: We can multiply two n bit integers in $O(n^{1.585})$ time.

Example 3: Matrix multiplication:

- Assume we are given two $n \times n$ matrix X and Y to multiply.
- These are huge matrices, say $n \approx 50,000$.
- The native algorithm: traverse each row i of X and each column j of Y (n^2 choices) and compute $\sum_{k=1}^{n} X_{i,k} \cdot Y_{k,j}$ ($O(n)$ multiplications per coordinate).
- Total time will be $O(n^3)$.
- Want to use divide and conquer to speed things up.
- For simplicity assume n is a power of 2.
- Break each of X and Y into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} I & J \\ K & L \end{bmatrix}}_{Z}$$

Example 3: Matrix multiplication:

- Divide and conquer.
- Therefore:

$$\left.\begin{array}{l} I = AE + BG \\ J = AF + BH \\ K = CE + DG \\ L = CF + DH \end{array}\right\} \longrightarrow \text{need 8 multiplications of } \tfrac{n}{2} \times \tfrac{n}{2} \text{ submatrices.}$$

- We also need to spend $O(n^2)$ time to add up these results.
- If $T(n)$ is the time to multiply two matrices of size $n \times n$ each, then:

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

- Using master theorem: $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.
- So this is as bad as the naive algorithm. No improvement yet.
- We use an idea similar to the one for multiplication of large integers: reduce the number of subproblems using a clever trick.

Matrix multiplication — Strassen's Algorithm (cont'd):

- Compute the following 7 multiplications (each consisting of two subproblems of size $\frac{n}{2}$ each):

$$
\begin{aligned}
S_1 &= A(F - H)\\
S_2 &= (A + B)H\\
S_3 &= (C + D)E\\
S_4 &= D(G - E)\\
S_5 &= (A + D)(E + H)\\
S_6 &= (B - D)(G + H)\\
S_7 &= (A - C)(E + F)
\end{aligned}
$$

- Then:

$$
\begin{aligned}
I &= S_5 + S_6 + S_4 - S_2\\
&= (A + D)(E + H) + (B - D)(G + H) + D(G - E) - (A + B)H\\
&= AE + DE + AH + DH + BG - DG + BH - DH +\\
&\qquad DG - DE - AH - BH\\
&= AE + BG
\end{aligned}
$$

Matrix multiplication (cont'd):

- Similarly, it can be verified easily that:

$$\begin{aligned} J &= S_1 + S_2 \\ K &= S_3 + S_4 \\ L &= S_1 - S_7 - S_3 + S_5 \end{aligned}$$

- (No, I do not expect you to remember by heart the different terms and additions.)
- So to compute I, J, K, and L, we only need to compute $S_1, \dots, S_7$; this requires solving seven subproblems of size $\frac{n}{2}$, plus a constant (at most 16) number of addition each taking $O(n^2)$ time.

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

- Using master theorem and since $\log_2 7 \approx 2.808$:

$$T(n) \in O(n^{2.808})$$

- Matrix multiplication is still an active research topic to this day.
  - Current best algorithm [V14] is $O(n^\omega)$ for $\omega = 2.3728...$
  - For $n = 60,000$: $n^3 \approx 2 \cdot 10^{14}$ and $n^{2.3728} \approx 2 \cdot 10^{11}$;
    $\Rightarrow$ this algorithm is about 1,000 times faster than the naive algorithm.
  - Still open — can we get $O(n^{2+\epsilon})$ for any $\epsilon > 0$?

Summary for Divide-and-Conquer:

- We think of recursion as "solve the problem for instance of size n assuming that a subinstance of size $n-1$ is already solved."
  That should be your initial approach.
- But after the initial recurssion, try the Divide-and-Conquer approach (multiple recursive calls on much smaller subinstances), which might substantially improve runtime:
  - break that input of size n to multiple subinstances (e.g., two subinstances of size $\frac{n}{2}$, three subinstances of size $\frac{n}{3}$, or several subinstances of different size)
  - solve each subproblem recursively
  - leverage on the solved subinstances to solve the entire, size n, instance.
- And after the initial D&C design (especially when the run-time recurrence relation falls into Case 1 of Master Theorem) see if you can find clever tricks to reduce the number of recursive calls, at the expense of more (but not asymptotically more) non-recursive operations.