Agenda:

- Divide and Conquer Revisited
  - Exponentiation
  - Karatsuba's algorithm for multiplying large integers (not in CLRS but widely available for its notoriety)
  - Strassen's algorithm for matrix multiplication (CLRS Ch.4.2)
- Introduction to Dynamic Programming (CLRS p357-369)

The Activity Selection Problem

- The input: n jobs, each with a start time s_i and finish time f_i; $0 \leq s_i < f_i$.
- All jobs require a certain resource (classes that require a hall)
- Two jobs i and j are said to conflict (or are incompatible) if the two intervals $[s_i, f_i)$ and $[s_j, f_j)$ overlap. (Only one class can be taught in a hall at any given moment.)
- Goal: schedule a largest subset of non-conflicting jobs.

- Note: a largest subset, not the largest subset — there could be more than one optimal solution.

i	1	2	3	4
s_i	1	3	1	4
f_i	3	5	4	5

$\{1, 2\}$, $\{1, 4\}$, $\{3, 4\}$ are all optimal solutions.

- How can we design an algorithm that *always* finds an optimal set of non-conflicting jobs?

Activity Selection Problem

- First attempt: a solution by exhaustively trying all possibilities.
- For each of the first choice $\langle s_i, f_i \rangle$
  1. remove all jobs conflicting with i and
  2. recurse on remaining jobs.
- After removing job i and all jobs conflicting with i, we
  1. have the same type of input as the original problem, and need to produce the same type of output
  2. we look for an optimal solution for this subproblem, since the overall solution must be $\{i\} \cup OPT(\text{subproblem})$.
- OK. let's make this recursion into an algorithm

Activity Selection Problem - First Attempt

- Here is the algorithm

```
procedure FirstAttempt(J)
** J an array of size n of jobs, J[i] is a tuple ⟨s_i, f_i⟩.
A ←a new array of size n
for (i ← 1 to n) do
    J_i ← all jobs not-conflicting with job i
    A[i] ← 1+ FirstAttempt(J_i)
                ** max-set of non-conflicting jobs in this subproblem
return max(A) (or 0 if A is empty)
```

- This algorithm is very slow
  - In each step we make $|J|$ possible choices.
  - Suppose each job conflicts with at most c other jobs, then total # of recursive calls is at least
    $n \cdot (n-(c+1)) \cdot (n-2(c+1)) \cdot (n-3(c+1)) \cdot \ldots \cdot 1$ — exponential!

Activity Selection Problem — Designing a Greedy Algorithm

- Let's design a greedy algorithm.
- The challenge: there are too many ways to make a "first choice".
- 1st attempt: suppose we sort the jobs based on start time, i.e. $s_1 \leq s_2 \leq \ldots \leq s_n$. Schedule a job if we can.
  - Bad example: $s_1 = 1$, $s_2 = 2$, $s_3 = 3$
    $f_1 = 4$, $f_2 = 3$, $f_3 = 4$
    The greedy can only schedule job 1 but the optimal is to schedule 2 and 3.
- 2nd attempt: Try to schedule shorter jobs first, i.e. sort jobs based on $f_i - s_i$ in non-decreasing order and then Schedule a job if we can.
  - Bad example: $s_1 = 3$, $s_2 = 1$, $s_4 = 4$
    $f_1 = 5$, $f_2 = 4$, $f_3 = 8$
    The greedy can only schedule job 1 but the optimal is to schedule 2 and 3.
- 3rd attempt: Sort the jobs based on non-decreasing order of finish time, i.e. $f_1 \leq f_2 \leq \ldots \leq f_n$ and then Schedule a job if we can.

Activity Selection Problem — Designing a Greedy Algorithm

- Why "earliest finish time"?
- Because there's *always* an optimal solution that contains the job with the earliest finish time. Let's argue for this.
  - Denote $\langle s_1, f_1 \rangle$ as a job with the earliest finish time.
  - Take an optimal set A of non-conflicting jobs.
  - The key observation: there's always a job j in A which we can swap with job 1.
    - A is a maximal solution not containing job 1. Hence, $A \cup \{1\}$ has conflicts.
    - Which jobs in A can conflict with job 1?
    - Only the first job in A (that start the earliest and ends the earliest): all other jobs start AFTER the first job ends which is AFTER job 1 ends.
    - Swap out the first job in A and swap in job 1 — the result is a non-conflicting set of size the same as A, so it is a maximal non-conflicting set.

Correctness:

```
procedure Activity-Selection (S)

   Sort activities in S s.t.  f_1 ≤ f_2 ≤ ... ≤ f_n
   A ← ∅; e ← 0;
   for i ← 1 to n do
      if s_i ≥ e then
         A ← A ∪ {i}
         e ← f_i
   return A
```

- Note that the running time of this algorithm is $\Theta(n \log n)$ as it's the time needed to sort the jobs and each iteration of the loop takes constant time.
- In this algorithm A is the set of jobs scheduled by our algorithm and e at any given time is the time at which the last scheduled job finishes (i.e. the earliest time we can schedule the next job).
- We prove that this 3rd attempt is correct.
- First note that, since we always schedule a new job if its start time s_i is larger than the finish time of the last scheduled job we get a set of compatible jobs at the end.

When Greedy Algorithms Work

- To complete correctness, we have to show our algorithm obtains an optimum solution.
- Promising: we say a schedule after step i is promising if it can be extended to an optimum schedule using a subset of jobs in $\{i+1, \ldots, n\}$.
- Note that this means that there is an optimum solution such that the decisions we have made for the first i jobs are consistant with that optimum (if we scheduled a job optimum has it too and if decided not to schedule it optimum doesn't have it either).
- We prove that after every step i, the partial solution we have is promising.
- Formally, let A_i and e_i denote the values of A and e after iteration i of the for-loop, respectively.
- Note that $A_0 = \emptyset$ and $e = 0$.
- We show that after each iteration $i \geq 0$, the decisions we have made so far are all correct in the sense that there is an optimum solution which contains everything we have selected so far and everything that we have decided not to include (so far) does not belong to that optimum either:
  Lemma: For every $0 \leq i \leq n$, there is an optimal solution A_{opt} such that $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$.

- Note that if we prove this for all $0 \leq i \leq n$, and in particular for $i = n$ we have: $A_n \subseteq A_{opt} \subseteq A_n \cup \emptyset$ which implies $A_n = A_{opt}$, i.e. our solution is the same as some optimum solution; this is what we wanted.
- We use induction on i to prove the above lemma.
- Base: $i = 0$, clearly empty schedule can be extended to an optimal one from $\{1, \ldots, n\}$; Thus $\emptyset = A_0 \subseteq A_{opt} \subseteq \emptyset \cup \{1, \ldots, n\}$.
- Induction Step: Suppose we have a promising schedule after step $i \geq$, i.e. there is an optimum solution A_{opt} such that: $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$.
- We consider three cases
- **Case 1:** if $s_{i+1} < e_i$ so we cannot schedule job $i+1$ because it overlaps with one of the previously scheduled ones; i.e. $A_{i+1} = A_i$. Since $A_i \subseteq A_{opt}$, A_{opt} has the job from A_i that is overlapping with $i+1$ as well, so A_{opt} cannot have $i+1$ either. Thus: $A_{i+1} = A_i \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \ldots, n\}$.
- **Case 2:** $s_{i+1} \geq e_i$; so $A_{i+1} = A_i \cup \{i+1\}$.

  Case 2A: This is the easy case where $i+1 \in A_{opt}$ then $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \ldots, n\}$ and we are done.

  Case 2B: $i+1 \notin A_{opt}$;

- Case 2B: $i+1 \notin A_{opt}$;
  - Note that $s_{i+1} \geq e_i$ and so $i+1$ does not conflict with anything in A_i;
  - On the other hand we cannot add $i+1$ to A_{opt} (otherwise it would have not been an optimum solution); so there must be a job $j \in A_{opt}$ conflicting with $i+1$. Since $A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$ and since $i+1$ does not conflict with anything in A_i: $j \geq i+2$.
  - Let $j \geq i+2$ be a job of A_{opt} with the earliest finish time that is conflicting with $i+1$.
  - All activities in $A_{opt} - A_i - \{j\}$ have start-time after f_j (or they will overlap with job j); so they do not overlap with job $i+1$ because $i+1$ finishes before f_j.
  - So $A'_{opt} = (A_{opt} - \{j\}) \cup \{i+1\}$ is feasible and has the same size as A_{opt}; i.e. it is another optimum solution and $A_{i+1} \subseteq A'_{opt} \subseteq A_{i+1} \cup \{i+2, \ldots, n\}$.

Divide and Conquer Revisited

- We have seen an example of *divide-and-conquer*, the merge sort.
- 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.)
- Here, we will see examples of some smart ways to play with divide-and-conquer to come up with more efficient algorithms.

Divide and Conquer and More!

- There are cases 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: $a = 1$, $b = 2$, $f(n) = 5 \in \Theta(n^0 \left(\log(n)\right)^0)$ so case 2 applies and we get $T(n) = \Theta(n^0 \left(\log(n)\right)^1) = \Theta(\log(n))$
- 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) in CMPUT 272.
- 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 $\texttt{exp}(b, \frac{n}{2})$, and squaring it.
  For odd n — $\texttt{exp}(b, \lceil \frac{n}{2} \rceil) = b \times \texttt{exp}(b, \lfloor \frac{n}{2} \rfloor)$, so save $\texttt{exp}(b, \lfloor \frac{n}{2} \rfloor)$, square it, and make one more multiplication with b.
- Note: 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): $b^{25} \cdot b^{25}$; $b \cdot b^{24}$; $b^{12} \cdot b^{12}$; $b^6 \cdot b^6$; $b^3 \cdot b^3$; $b \cdot b^2$; $b \cdot b$
- `procedure Power(`b, n`)`

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

Example 1: Exponentiation

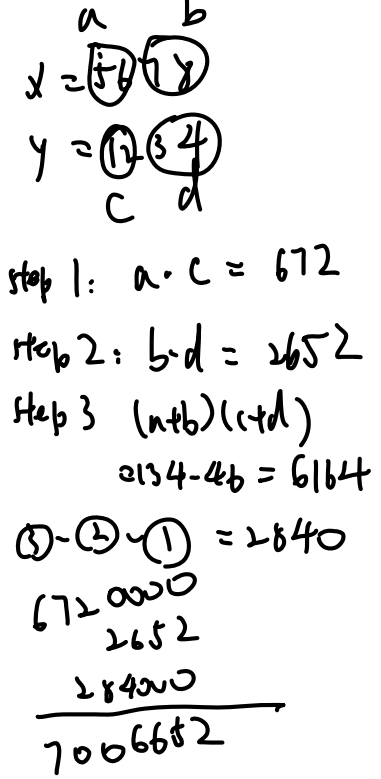
- `Procedure Power`(b, n)
  `if` $(n = 0)$ `then`
    `return` 1
  `else`
    `if` n `is odd then`
      $a \leftarrow$ `Power`$(b, n-1)$
      `return` $a \cdot b$
    `else`
      $a \leftarrow$ `Power`$(b, n/2)$
      `return` $a \cdot a$
- Let $T(n)$ be the number of multiplications required to compute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

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

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

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

- 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 (most significant bits), x denotes the $\frac{n}{2}$ LSBs (least significant bits).

$$I = \boxed{w}\boxed{x}$$

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

$$J = \boxed{y}\boxed{z}$$

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

Karatsuba's Algorithm for Multiplying Large Integers

- From above, $I \cdot J = w \cdot y \cdot 2^n + (w \cdot z + x \cdot y)2^{n/2} + x \cdot z$.
- To multiply by 2^n only needs to shift-left n bits; each shiftleft 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: $T(n) = 4T(\frac{n}{2}) + O(n)$.
- Using master theorem: $T(n) \in \Theta(n^2)$.
- But this is not better than the naive algorithm!! What should we do?
- 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 $r = (w + x)(y + z) = w \cdot y + (w \cdot z + x \cdot y) + x \cdot y$.
- So r contains all the 4 terms we need to compute $I \cdot J$, but not individually.
- What if we compute $p = w \cdot y$ and $q = x \cdot y$, too? Then we have:
  - $(w \cdot z + x \cdot y) = r - p - q$
  - $w \cdot y = p$
  - $x \cdot y = q$
- 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:

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

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 $S_1, \ldots, S_7$; 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 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.

Dynamic programming introduction:

- An algorithm design paradigm
- Usually for optimization problems
- Typically like divide-and-conquer uses solutions to subproblems to solve the problem, BUT
- Key idea: *Avoids re-computation*
- **Example— Fibonacci numbers**

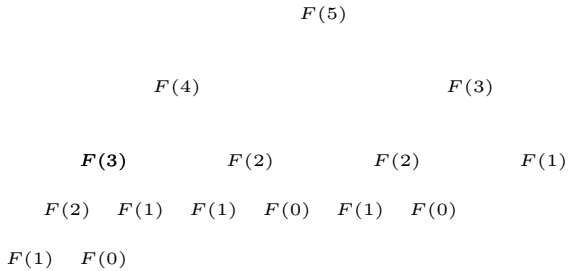
$$F(n) = \begin{cases} n, & \text{when } n = 0, 1 \\ F(n-1) + F(n-2), & \text{when } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9
$F(n)$	0	1	1	2	3	5	8	13	21	34

- Question: how do we compute $F(n)$?

1st Naive Fibonacci implementation – recursion

- Recursion tree for a direct implementation by a recursive function



- A lot of repeated function calls
- Running time recurrence

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

- Exponential running time: $T(n) \in \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

2^{nd} Fibonacci implementation – Dynamic Programming

- Idea: Do the computation in a bottom up manner; store the compute values for future use.
- Define array $A[0..n]$ where $A[i]$ is going to store $F(i)$.
- Fill in the values of $A[0]$ and $A[1]$ from the definition (initialization)
- Start computing $A[i]$, from $i = 2$ onward, using the recurrence

$$A[i] \longleftarrow A[i-1] + A[i-2]$$

- Pseudocode:

```
procedure DynFib(n)
A[0] ← 0
A[1] ← 1
for j ← 2 to n do
    A[j] ← A[j − 1] + A[j − 2]
return A[n]
```

- Running time $T(n) \in \Theta(n)$

General steps in designing a dynamic programming solution:

1. Find a recurrence relation for the problem.

2. Check to see if the recurrence repeatedly makes the same calls, and if all of the recursive calls live in a moderately sized universe.
   In Fibonacci, we make a lot of calls, but only to n possible values: $F(0), F(1), F(2), ..., F(n-1)$.

3. Describe an array of values that you want to compute. Each cell in the array will be the result of a possible recursive call, and the value of the solution for the appropriate subproblem.
   $A[i]$ stores the value $F(i)$.

4. Fill the array bottom-up: from the cells corresponding to the base case of the recursion, to cells we now can compute.
   First fill $A[0], A[1]$, then $A[2]$, then $A[3]$,..., then $A[n]$.

5. Extract the solution from the array.
   $A[n]$.

Rod cutting

- We have a rod of length n to sell, the selling price of a piece of length i is p_i (given).
- We want to cut the rod into pieces and sell those to get a maximum profit.
- Question: into what size pieces should we cut the rod?
- Example: $n = 5$, $p_1 = 2$, $p_2 = 6$, $p_3 = 7$, $p_4 = 11$, and $p_5 = 13$ best way would be to cut into $2 + 2 + 1$.
- (naive) solution: try all possible combinations; we have two options of cutting or not cutting at each distance i from the left end: 2^n possible ways of cutting the rod

2nd try: Recursive

- `procedure Cut-Rod` (n, p)

```
if n = 0 then
    return 0
else
    q = −∞
    for i ← 1 to n do
        q = max{q, p_i + Cut-Rod(n − i, p)}
    return q
```

- Problem is that we will might be solving some sub-problems multiple times!
- e.g. if we first cut a piece of length 1 and then a piece of length 3 we will have to solve the problem for a rod of length $n-4$ optimally; this happens again when in the first two iterations we cut two pieces of length 2;
- It turns out the time complexity of this algorithm is still exponential in n:

$$T(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 + \sum_{i=1}^{n-1} T(i) & \text{for } n \geq 1 \end{cases}$$

- It can be shown that $T(n) \in \Theta(2^n)$.

Dynamic Programming

- What if we compute the solution to each subproblem only ONCE and store them into a table? whenever we need that solution we simply look-up at the table.
- In the first implementation, we use the same recursive method but have a table $r[0...n]$, and we store the optimum solution of cutting a rod of length i in $r[i]$ the first time we compute it.
- whenever we need to make a recursive call to $Cut - Rod(i, p)$ we check if $r[i]$ is already computed or not and if so we return that value.

- ```
  procedure Cut-Rod (n, p)
      for i = 0 to n do
          r[i] = −∞
      return Memoized-Cut-Rod (n, p)

  procedure Memoized-Cut-Rod (n, p)
      if r[n] ≥ 0 then
          reutnr r[n]
      if n = 0 then
          return 0
      else
          q = −∞
          for i ← 1 to n do
              q = max{q, p_i + Memoized-Cut-Rod(n − i, p)}
          r[n] = q
          return q
  ```

- It can be seen that each entry $r[i]$ is computed only once.
  ```