Algorithms Analysis and Design

Chapter 7

Dynamic Programming
Part 1

Introduction

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs

- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

It is well-known that there is no universal technique that can be the best-performing for all problems

Dynamic Programming

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

"**Programming**" refers to a tabular method not writing code

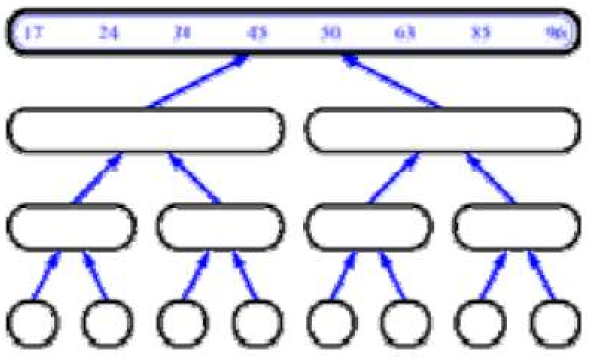
- An algorithm design technique.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- Used when problem breaks down into recurring small subproblems.
- Dynamic programming is typically applied to optimization problems.
    - In such problem there can be many solutions. Each solution has a value, and we wish to find a solution with the optimal value.

Dynamic Programming

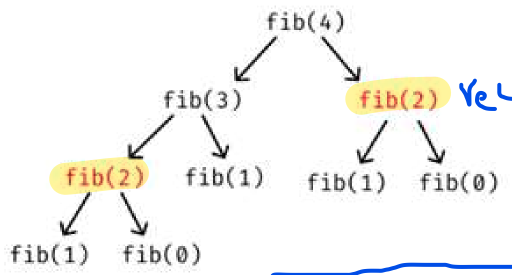
- Main Idea
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once.
  - record solutions in a table .
  - extract solution to the initial instance from that table.

Dynamic Programming vs Divide and Conquer

- Dynamic programming is a design technique, like divide and conquer.
- Divide-and-Conquer (DAC): subproblems are independent.
- Dynamic Programming (DP): subproblems are not independent.
- For example: in MergeSort, the subproblems are independent (all different).



Subproblems are **independent**



The larger Fibonnaci sequence has overlapping function call.

إذا بدي أحطها recurcive بتكلف exponintial لـ

لانه كل ما بحل

بدي احطها اكثر من مرة

Subproblems are **dependent (overlapping)**

Dynamic Programming vs Divide and Conquer

- In solving problems with overlapping subproblems
  - A DAC algorithm does redundant work (inefficient)
    - Repeatedly solves common subproblems
  - A DP algorithm solves each problem just once (more efficient)
    - Saves its result in a table

Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms

By "inefficient", we mean that the same recursive call is made over and over

Optimization Problems

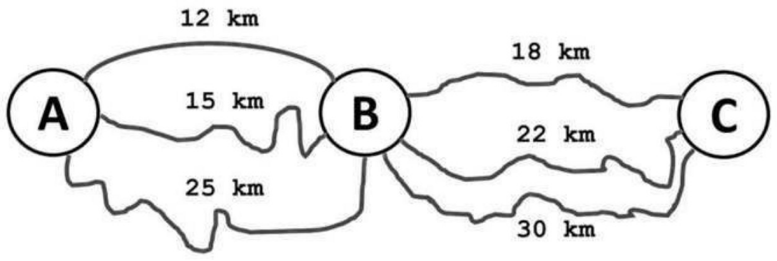
- DP typically applied to optimization problems
- In an optimization problem
  - There are many possible solutions (feasible solutions)
  - Each solution has a value
  - Want to find an optimal solution to the problem
    - A solution with the optimal value (min or max value)
  - Wrong to say "the" optimal solution to the problem
    - There may be several solutions with the same optimal value

Elements of Dynamic Programming

- When should we look for a DP solution to an optimization problem?
- Two key ingredients for the problem
  - **Optimal substructure**
    - An optimal solution to a problem contains within it optimal solutions to subproblems.
  - **Overlapping subproblems**
    - When a recursive algorithm revisits the same problem over and over again, we say that the problem has overlapping subproblems.

Optimal substructure

- A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.
- This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.
- For example, the **Shortest Path problem** has the following optimal substructure property:
  - If a node B lies in the shortest path from a source node A to destination node C then the shortest path from A to C is a combination of the shortest path from A to B and the shortest path from B to C.

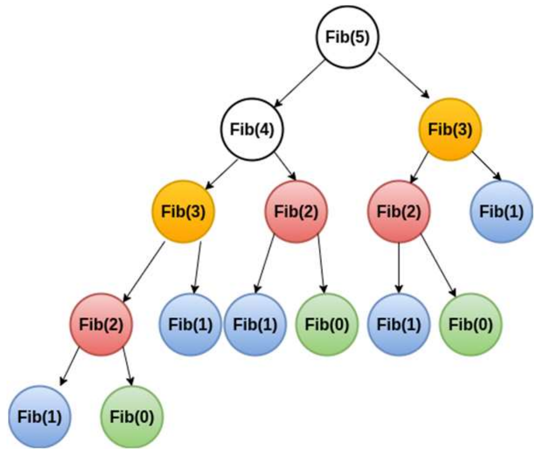


wavy lines indicate shortest paths, i.e., there might be other vertices that are not shown here

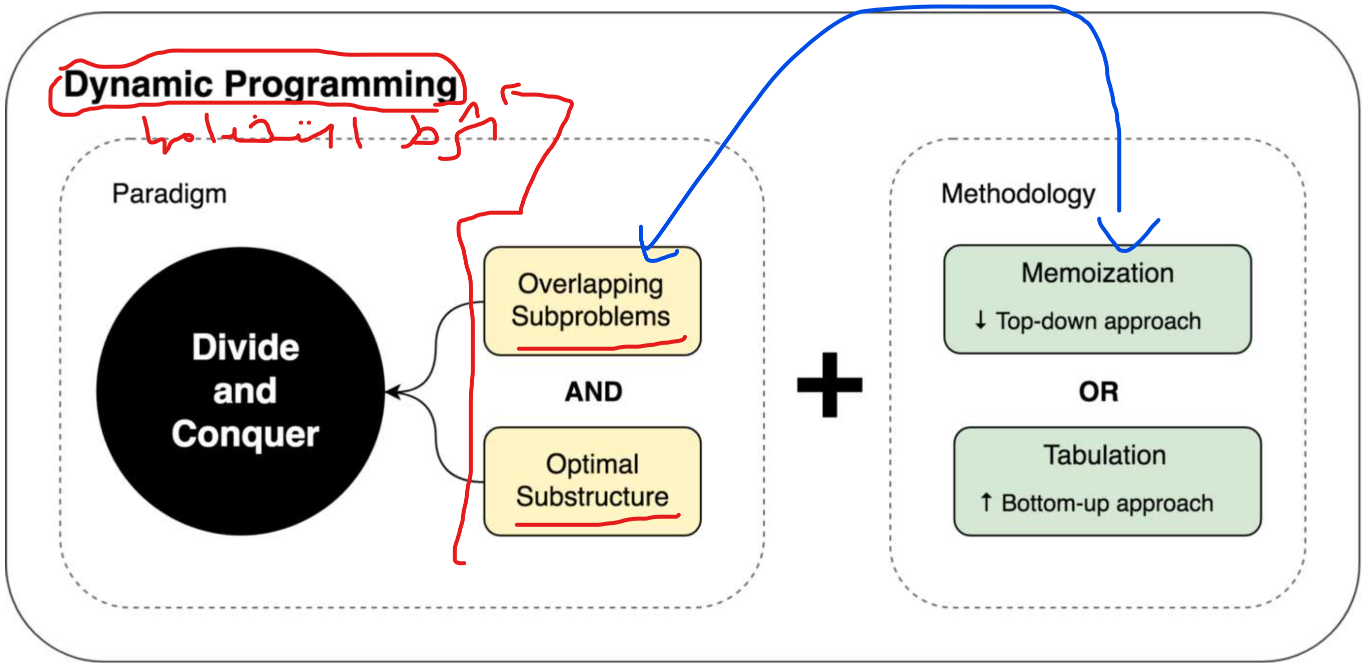
Overlapping Subproblems

- Dynamic Programming algorithms typically take advantage of overlapping subproblems
  - By solving each problem once.
  - Then storing the solutions in a table where it can be looked up when needed.
  - Using constant time per lookup.
- Example: If we look at fib(5), we can see the repeated calculations:

Dynamic programming is applicable when the subproblems are dependent



Difference Between DP and DC After All



If same subproblem is solved several times, we can use table to store result of a subproblem the first time it is computed and thus never have to recompute it again

Development of a DP Algorithm

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from the information computed in Step 3

Applications of the DP Strategy

In this course we will discuss some selected problems that can be efficiently solved using DP technique

- Fibonacci sequence.
- Longest Common Subsequence (LCS) problem.
- Edit distance (Levenshtein distance)
- 0/1 Knapsack problem.
- Coin-row problem.
- Coin change-making problem.
- Matrix-chain multiplication

Fibonacci numbers sequence

Fibonacci sequence

- The Fibonacci Sequence is the series of numbers that starts with two ones at the beginning, and then by a series of steadily increasing numbers. The sequence follows the rule that each number is equal to the sum of the preceding two numbers.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

- In mathematical terms, the sequence F(n) of Fibonacci numbers is defined by the recurrence relation:

$$F(n) = F(n-1) + F(n-2), \qquad n > 2.$$

$$F(1) = 1, F(2) = 1 \qquad\qquad (Base\ case)$$

Fibonacci sequence

- Let's we are trying to solve the problem of finding the nth Fibonacci number.
- We will write an algorithm called *fib(n)* that takes an integer positive number n and finds the nth Fibonacci number. For example, if the given number n is five, we want to be able to find the fifth Fibonacci number which is 5.

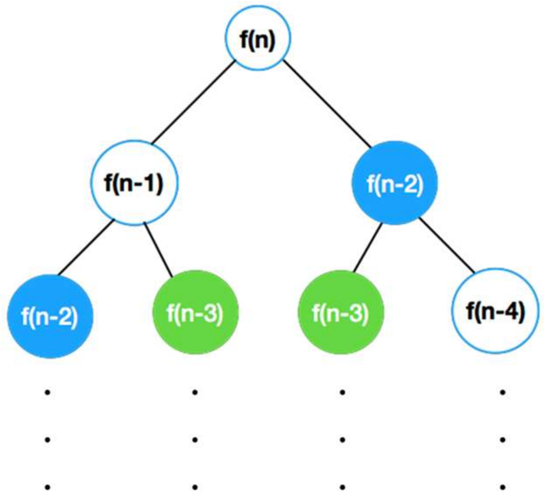
- **We will solve the problem using 3 methods:**

  1. Top-down recursive approach.
  2. Top-down with memoization.
  3. Bottom-up dynamic programming (Tabulation).

Top-down recursive approach

- To compute $fib(n)$ in the recursive approach, we first try to find the solutions to $fib(n-1)$ and $fib(n-2)$. But to find $fib(n-1)$, we need to find $fib(n-2)$ and $fib(n-3)$. This continues until we reach the base cases: fib(1) and fib(2).

```
ALGORITHM fib ( n )
{
    If (n == 1 or n==2)
      result = 1
    else
      result = fib (n – 1) + fib (n-2)
    return result
}
```



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

$$T(n) = \begin{cases} d, & n = 1, 2 \\ T(n-1) + T(n-2) + c, & n > 2 \end{cases}$$

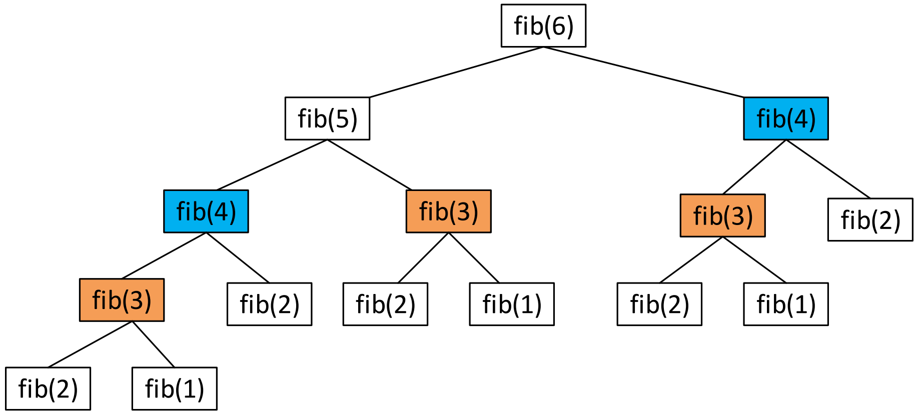
T(n) ∈ $O(1.6^n)$ **exponential time**

Top-down recursive approach

- This algorithm is **inefficient**. Same subproblems are solved many times.

A lot of repeated computations

What if the value of n increases?
50 , 100 ??



- Solution → Top-down **Memoization and bottom-up DP**

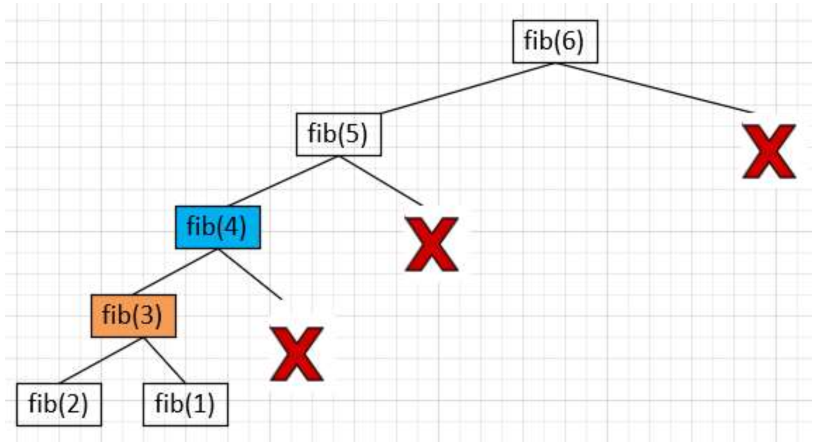
Top-down memoization approach

- **Store** some of the intermediate results, so we don't have to repeat some computation

```
ALGORITHM fib ( n , M )
{
    if M[n] != -I          // or != null
        return M[n]
    If (n == 1 or n==2)
        result = 1
    else
        result = fib (n – 1) + fib (n-2)
    M[n] = result          // store
    return result
}
```

M Use an array to store the solution for each solved subproblem

-1	-1	-1	-1	.....	-1



More efficient than previous algorithm, but still a recursive algorithm

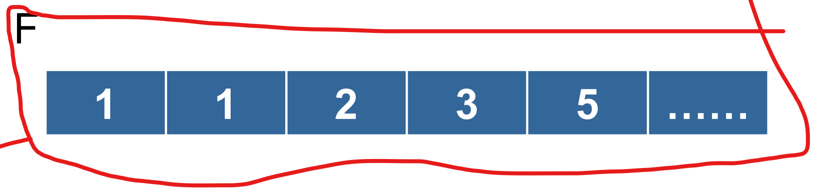
Can improve it further????

T(n) ∈ $O(n)$ **Linear time**

Bottom-up approach

- In the bottom-up dynamic programming approach, we'll reorganize the order in which we solve the subproblems.
- We'll compute F(1), then F(2), then F(3), and so on:

```
ALGORITHM fib ( n , M )
{
    Create array F
    F[1] = 1
    F[2] = 1
    for i = 3 to n
        F[i] = F[i- 1] + F[i-2]
    return F[n]
}
```



T(n) ∈ $O(n)$ **Linear time**

Space complexity??

Exercise: Can we improve this algorithm more???
Are we need to store all intermediate results in an array?

Exercise: Can we improve the previous algorithm???
Are we need to store all intermediate results in an array?

- The idea is to use a **few variables** to keep track of the last two Fibonacci numbers and update them as you progress through the sequence
- This method is more space-efficient than using a table as it only requires constant space

```
ALGORITHM fibonacci ( n )
{
    // check if n<=1
    firstNumber = 1
    secondNumber = 1
    for i from 3 to n
        nextNumber = firstNumber + secondNumber
        firstNumber = secondNumber
        secondNumber = nextNumber

    return secondNumber
}
```

T(n) ∈ $O(n)$ **Linear time**

More space-efficient O(1)

Exercise: Sum of Fibonacci Numbers

- Given a positive number ***n***, find value of f0 + f1 + f2 + …. + fn where fi indicates i'th Fibonacci number.

  Remember that f0 = 0, f1 = 1, f2 = 1, f3 = 2, f4 = 3, f5 = 5, …….. Fn=fn-1 + fn-2

- ***Input : n = 3***
  Output : 4
  Explanation : 0 + 1 + 1 + 2 = 4
- ***Input : n = 4***
  Output : 7
  Explanation : 0 + 1 + 1 + 2 + 3 = 7

Key notes:

$\star$

- **A problem can be solved using dynamic programming if**

(1) it has **optimal substructure**

(2) it has **overlapping subproblems**

- Therefore, any recursive problem having overlapped sub-problems can be solved using dynamic programming.
- In Fibonacci numbers, if we will use **recursion** many sub-problem are calculated more than once which is inefficient and we get **exponential time complexity.**
- What can be done here is Using **dynamic programming**, we can calculate the subproblem solution once and store it for further reference. This will reduce the time complexity to **linear** in this case.

Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later.

Top Down Recursive vs Bottom Up Dynamic Programming

- **Bottom Up DP algorithms:** *
  - More efficient.
  - Regular pattern of table access can be exploited to reduce time or space.
  - Take advantage of the overlapping-subproblems property.
- **Top-down recursive algorithm:**
  - Repeatedly resolve each subproblem each time it appears in the recursion tree.
  - Recursion overhead.
  - Only work when the total number of subproblems in the recursion is small.

Memoization

$\ast$

- **A variation of dynamic programming**
  - Offers the efficiency of the usual dynamic programming approach while maintaining a top-down strategy
- **Ideas**
  - A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem
  - Initially contain a **special value** to indicate that the entry has yet to be filled in. when the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table.

If you notice that there are a lot of repeated computations, you can store some of the intermediate results so that you don't have to repeat those computations. This process is called **memorization**

Top Down Memoization vs Bottom Up DP

- **Bottom-up DP** *
  - Regular pattern of table access can be exploited to reduce time or space.
- **Top-down + memoization**
  - solve only subproblems that are definitely required.
  - Recursion overhead.

Both methods take advantage of the overlapping subproblems property

Comparison

*

Top-Down	Top-Down Memoization	Bottom-UP DP

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen’s algorithm
  - Exponentiation problem
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Exponentiation problem

❑ The divide-and-conquer technique can be successfully applied to handle the exponentiation problem

❑ Recall that we solved the same problem in linear time [$\theta\ (n)$] using two Conventional methods:

- Brute force
- Simple recursive algorithm / Decrease and conquer [**unbalanced** partitioning]

Exponentiation problem

- Problem: exponentiation problem [Computing a^n (a > 0, n a nonnegative integer)]

$$a^n = a * \underbrace{a * a \ldots\ldots\ldots\ldots *}_{\text{n times}} a$$

- Time complexity: $\theta\ (n)$

- **Can we find a better (faster) algorithm?**

Naive algorithm

```
ALGORITHM pow ( a , n )
{
    result = 1
    for $i$ = 1 to n
        result = result * a
    return result
}
```

Decrease and conquer for solving a^n

$$a^n = a * \overset{a^{n-1}}{\underbrace{\boxed{a * a * a \ldots\ldots\ldots * a}}_{n-1}}$$

$$a^n = a * a^{n-1}$$

1. Decrease the problem by **1**, and

   solve the smaller instance of size n − 1 recursively.

1. Derive a recurrence relation and solve it.
2. Find the time complexity.
3. Is it a good algorithm for solving this problem?

$$a^n = a * a^{n-1}$$
$$= a * a * a^{n-2}$$

Recursive algorithm

```
ALGORITHM Rec_pow ( a , n )
{


}
```

Divide and conquer approach for solving a^n

a^8 = a * a * a * a * a * a * a * a

= $a^4 \cdot a^4$

$a^n = a^{n/2} \cdot a^{n/2}$

n is even

a^9 = a * a^8

= a * $a^4 \cdot a^4$

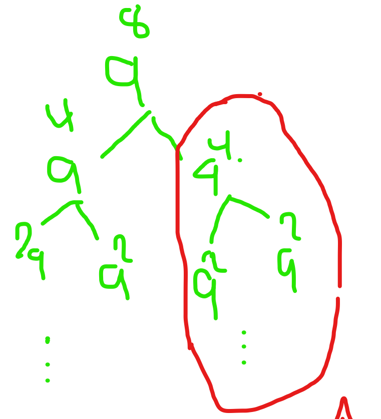
a^n = a * $a^{(n-1)/2} \cdot a^{(n-1)/2}$

n is odd

1. Divide and conquer approach is achieved by creating sub problems of size n/2 [**Balanced partitioning**]
2. Each subproblem is solved recursively until n = 1 [**base case**]
3. The recursive solution:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \textit{if } n \textit{ is even} \\ a * a^{(n-1)/2} \cdot a^{(n-1)/2} & \textit{if } n \textit{ is odd} \end{cases}$$

Pseudocode of DAC approach



```
ALGORITHM $Pow\_DAC\ (a, n)$
{
        $\textbf{if}\ (n = 1)\ \textbf{then}$
            $return\ a$
        else
            $\textbf{return}\ Pow\_DAC(a, n/2) * Pow\_DAC(a, n/2)$
}
```

$T(\frac{n}{2}) + T(\frac{n}{2})$ calls.

- **Note**: This algorithm does not check whether n is even or odd

- **Overlapping problem: same subproblem solved two times**

- Complexity analysis:

  T(n) = 2 T(n/2) + C   n > 1

  using master theorem, T(n) ∈ O(n)

How does this algorithm compare with the conventional approach??

Improved DAC approach

Dynamic

```
ALGORITHM $Pow\_DAC\ (\overset{4}{a}, \overset{8}{n})$          8
{                                                                4
        $\textbf{if}\ (n = 1)\ \textbf{then}$
            $return\ a$                                   $y = 4^4$
        else
            $\textbf{y} = Pow\_DAC(a, n/2)$ ⟶ $T(n/2)$
            $\textbf{return}\ y * y$ ⟶ $O(1)$
                        $4^4 * 4^4$
}
```

- **To handle overlapping problem:**

Solve one subproblem and store its solution.

Multiply the partial solution by itself

This approach can be considered decrease by a half technique.

- Complexity analysis:

  T(n) = T(n/2) + C   n > 1

  using master theorem, T(n) ∈ **O(log n)**

This approach is asymptotically faster than previous methods.

Improved DAC approach

- **Rules to have an efficient Divide and Conquer method:**

1. **Balanced partitioning.**
2. **Subproblems are independent (no overlapping)**

Exercise

Modify the following pseudocode to handle both even and odd values of n

```
ALGORITHM $Pow\_DAC\ (a, n)$
{
        if $(n = 1)$ then
            return $a$
        else
            return $Pow\_DAC(a, n/2) * Pow\_DAC(a, n/2)$
}
```

$$a^n = \begin{cases} a^{n/2} \,.\, a^{n/2} & if\ n\ is\ even \\ \text{a} * a^{(n-1)/2} \,.\, a^{(n-1)/2} & if\ n\ is\ odd \end{cases}$$