Week 3

Solving Recurrences

1. Divide-and-Conquer
   - Merge Sort

2. Solving Recurrences
   - Recursion Trees
   - Master Theorem

3. Divide-and-Conquer
   - Matrix multiplication

4. Tutorial
General remarks

- First we continue with an important example for Divide-and-Conquer, namely **Merge Sort**.
- Then we present a basic tool for analysing algorithms by **Solving Recurrences**.
- We conclude by considering an example, namely **Matrix Multiplication**.

**Reading from CLRS for week 3**
- Chapter 4
Another example: Merge-Sort

A sorting algorithm based on divide and conquer. The worst-case running time has a lower order of growth than insertion sort.

Again we are dealing with subproblems of sorting subarrays $A[p \ldots q]$ Initially, $p = 1$ and $q = A.length$, but these values change again as we recurse through subproblems.

To sort $A[p \ldots q]$:

**Divide** by splitting into two subarrays $A[p \ldots r]$ and $A[r+1 \ldots q]$, where $r$ is the halfway point of $A[p \ldots q]$.

**Conquer** by recursively sorting the two subarrays $A[p \ldots r]$ and $A[r+1 \ldots q]$.

**Combine** by merging the two sorted subarrays $A[p \ldots r]$ and $A[r+1 \ldots q]$ to produce a single sorted subarray $A[p \ldots q]$.

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.
Another example: Merge-Sort

A sorting algorithm based on divide and conquer. The worst-case running time has a lower order of growth than insertion sort.

Again we are dealing with subproblems of sorting subarrays $A[p..q]$ Initially, $p = 1$ and $q = A.length$, but these values change again as we recurse through subproblems.

To sort $A[p..q]$:  

Divide by splitting into two subarrays $A[p..r]$ and $A[r+1..q]$, where $r$ is the halfway point of $A[p..q]$.

Conquer by recursively sorting the two subarrays $A[p..r]$ and $A[r+1..q]$.

Combine by merging the two sorted subarrays $A[p..r]$ and $A[r+1..q]$ to produce a single sorted subarray $A[p..q]$.

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.
Another example: Merge-Sort

A sorting algorithm based on divide and conquer. The worst-case running time has a lower order of growth than insertion sort.

Again we are dealing with subproblems of sorting subarrays $A[p..q]$ Initially, $p = 1$ and $q = A.length$, but these values change again as we recurse through subproblems.

To sort $A[p..q]$:

**Divide** by splitting into two subarrays $A[p..r]$ and $A[r+1..q]$, where $r$ is the halfway point of $A[p..q]$.

**Conquer** by recursively sorting the two subarrays $A[p..r]$ and $A[r+1..q]$.

**Combine** by merging the two sorted subarrays $A[p..r]$ and $A[r+1..q]$ to produce a single sorted subarray $A[p..q]$.

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.
Another example: Merge-Sort

\textbf{Merge-Sort}(A, p, q)

1. if \( p < q \) \quad // check for base case
2. \( r = \lfloor (p+q)/2 \rfloor \) \quad // divide
3. \textbf{Merge-Sort}(A, p, r) \quad // conquer
4. \textbf{Merge-Sort}(A, r+1, q) \quad // conquer
5. \textbf{Merge}(A, p, r, q) \quad // combine

Initial call: \textbf{Merge-Sort}(A, 1, A. length)
Merge

**Input:** Array $A$ and indices $p$, $r$, $q$ such that

- $p \leq r < q$
- Subarrays $A[p..r]$ and subarray $A[r+1..q]$ are sorted. By the restriction on $p$, $r$, $q$ neither subarray is empty.

**Output:** The two subarrays are merged into a single sorted subarray in $A[p..q]$.

We implement is so that it takes $\Theta(n)$ time, with $n = q - p + 1 = \text{the number of elements being merged.}$
**Merge**($A, p, r, q$)

1. $n_1 = r - p + 1$
2. $n_2 = q - r$
3. let $L[1..n_1+1]$ and $R[1..n_2+1]$ be new arrays
4. **for** $i = 1$ to $n_1$
   5. \[ L[i] = A[p+i-1] \]
5. **for** $j = 1$ to $n_2$
8. \[ L[n_1+1] = R[n_2+1] = \infty \]
9. $i = j = 1$
10. **for** $k = p$ to $q$
11. \[ \text{if } L[i] \leq R[j] \]
12. \[ A[k] = L[i] \]
13. \[ i = i + 1 \]
14. \[ \text{else } A[k] = R[j] \]
15. \[ j = j + 1 \]
Analysis of Merge-Sort

The runtime $T(n)$, where $n = q - p + 1 > 1$, satisfies:

$$T(n) = 2T(n/2) + \Theta(n).$$

We will show that $T(n) = \Theta(n \lg n)$.

- It can be shown (see tutorial-section) that $\Omega(n \lg n)$ comparisons are necessary in the worst case to sort $n$ numbers for any comparison-based algorithm: this is thus an (asymptotic) lower bound on the problem.
- Hence Merge-Sort is provably (asymptotically) optimal.
Analysing divide-and-conquer algorithms

Recall the divide-and-conquer paradigm:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

*Base case:* If the subproblem are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.

We use recurrences to characterise the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

A recurrence is a function defined in terms of

- one or more base cases, and
- itself, with smaller arguments
Analysing divide-and-conquer algorithms

Recall the divide-and-conquer paradigm:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

- *Base case*: If the subproblem are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.

We use **recurrences** to characterise the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

A recurrence is a function defined in terms of

- one or more base cases, and
- itself, with smaller arguments
Analysing divide-and-conquer algorithms

Recall the divide-and-conquer paradigm:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.
   *Base case:* If the subproblem are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.

We use **recurrences** to characterise the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

A **recurrence** is a function defined in terms of

- one or more base cases, and
- itself, with smaller arguments
Examples for recurrences

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n - 1) + 1 & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = n. \)

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = n \lg n + n. \)

- \( T(n) = \begin{cases} 
0 & \text{if } n = 2 \\
T(\sqrt{n}) + 1 & \text{if } n > 2 
\end{cases} \)

Solution: \( T(n) = \lg \lg n. \)

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n/3) + T(2n/3) + n & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = \Theta(n \lg n). \)
Examples for recurrences

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n - 1) + 1 & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = n \).

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = n \lg n + n \).

- \( T(n) = \begin{cases} 
0 & \text{if } n = 2 \\
T(\sqrt{n}) + 1 & \text{if } n > 2 
\end{cases} \)

Solution: \( T(n) = \lg \lg n \).

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n/3) + T(2n/3) + n & \text{if } n > 1 
\end{cases} \)

Solution: \( T(n) = \Theta(n \lg n) \).
Examples for recurrences

1. \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n - 1) + 1 & \text{if } n > 1
\end{cases} \)

Solution: \( T(n) = n \).

2. \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{if } n > 1
\end{cases} \)

Solution: \( T(n) = n \log n + n \).

3. \( T(n) = \begin{cases} 
0 & \text{if } n = 2 \\
T(\sqrt{n}) + 1 & \text{if } n > 2
\end{cases} \)

Solution: \( T(n) = \log \log n \).

4. \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n/3) + T(2n/3) + n & \text{if } n > 1
\end{cases} \)

Solution: \( T(n) = \Theta(n \log n) \).
Examples for recurrences

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n - 1) + 1 & \text{if } n > 1 
\end{cases} \)

  Solution: \( T(n) = n. \)

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{if } n > 1 
\end{cases} \)

  Solution: \( T(n) = n \log n + n. \)

- \( T(n) = \begin{cases} 
0 & \text{if } n = 2 \\
T(\sqrt{n}) + 1 & \text{if } n > 2 
\end{cases} \)

  Solution: \( T(n) = \log \log n. \)

- \( T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n/3) + T(2n/3) + n & \text{if } n > 1 
\end{cases} \)

  Solution: \( T(n) = \Theta(n \log n). \)
Main technical issues with recurrences

Floors and ceilings: The recurrence describing worst-case running time of Merge-Sort is really

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1
\end{cases}
\]

Exact vs. asymptotic functions Sometimes we are interested in the exact analysis of an algorithm (as for the Min-Max-Problem), at other times we are concerned with the asymptotic analysis (as for the Sorting Problem).

Boundary conditions Running time on small inputs is bounded by a constant: \( T(n) = \Theta(1) \) for small \( n \). We usually do not mention this constant, as it typically doesn’t change the order of growth of \( T(n) \). Such constants only play a role if we are interested in exact solutions.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions, as they usually do not matter.
Main technical issues with recurrences

Floors and ceilings: The recurrence describing worst-case running time of Merge-Sort is really

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1
\end{cases} \]

Exact vs. asymptotic functions Sometimes we are interested in the exact analysis of an algorithm (as for the Min-Max-Problem), at other times we are concerned with the asymptotic analysis (as for the Sorting Problem).

Boundary conditions Running time on small inputs is bounded by a constant: \( T(n) = \Theta(1) \) for small \( n \). We usually do not mention this constant, as it typically doesn’t change the order of growth of \( T(n) \). Such constants only play a role if we are interested in exact solutions.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions, as they usually do not matter.
Main technical issues with recurrences

Floors and ceilings: The recurrence describing worst-case running time of Merge-Sort is really

\[
T(n) = \begin{cases} 
    \Theta(1) & \text{if } n = 1 \\
    T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1
\end{cases}
\]

Exact vs. asymptotic functions Sometimes we are interested in the exact analysis of an algorithm (as for the Min-Max-Problem), at other times we are concerned with the asymptotic analysis (as for the Sorting Problem).

Boundary conditions Running time on small inputs is bounded by a constant: \( T(n) = \Theta(1) \) for small \( n \). We usually do not mention this constant, as it typically doesn’t change the order of growth of \( T(n) \). Such constants only play a role if we are interested in exact solutions.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions, as they usually do not matter.
Main technical issues with recurrences

Floors and ceilings: The recurrence describing worst-case running time of Merge-Sort is really

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1
\end{cases}
\]

Exact vs. asymptotic functions Sometimes we are interested in the exact analysis of an algorithm (as for the Min-Max-Problem), at other times we are concerned with the asymptotic analysis (as for the Sorting Problem).

Boundary conditions Running time on small inputs is bounded by a constant: \( T(n) = \Theta(1) \) for small \( n \). We usually do not mention this constant, as it typically doesn’t change the order of growth of \( T(n) \). Such constants only play a role if we are interested in exact solutions.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions, as they usually do not matter.
Recursion trees (quadratic growth)

Draw the unfolding of the recurrence

\[ T(n) = n + 4T(n/2). \]

We exploited that \( 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 = \Theta(2^k). \)
Recursion trees (quasi-linear and linear growth)

What about the “merge-sort” recurrence

\[ T(n) = n + 2T(n/2) \]

- Again the height of the tree is \( \lg n \).
- However now the “workload” of each level is equal to \( n \).

So here we get

\[ T(n) = \Theta(n \cdot \lg n). \]

And what about the recurrence

\[ T(n) = 1 + 2T(n/2) \]

- Again the height of the tree is \( \lg n \).
- The “workload” of the level is \( 1, 2, \ldots, 2^{\lg n} \).

So here we get

\[ T(n) = \Theta(n). \]
Master Theorem (simplified version)

Let \( a \geq 1 \) and \( b > 1 \) and \( c \geq 0 \) be constants.

Let \( T(n) \) be defined by the recurrence

\[
T(n) = aT(n/b) + \Theta(n^c),
\]

where \( n/b \) represents either \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \).

Then \( T(n) \) is bounded asymptotically as follows:

1. If \( c < \log_b a \) then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( c = \log_b a \) then \( T(n) = \Theta(n^c \lg n) \).
3. If \( c > \log_b a \) then \( T(n) = \Theta(n^c) \).

(General version: CLRS, Thm 4.1, p94.)
In other words

We need to give an equation for $T(n)$ of the form

$$T(n) = b^x \cdot T\left(\frac{n}{b}\right) + \Theta(n^c),$$

where the $x$ you have to find: $x = \log_b a$.

Then $T(n)$ is bounded asymptotically as follows:

1. If $x > c$ then $T(n) = \Theta(n^x)$.
2. If $x = c$ then $T(n) = \Theta(n^c \log n)$.
3. If $x < c$ then $T(n) = \Theta(n^c)$. 
Using the Master Theorem

- The runtime for **Min-Max** satisfies the recurrence:
  \[ T(n) = 2T(n/2) + \Theta(1). \]

  The Master Theorem (case 1) applies:
  \[ a = b = 2 \quad \text{and} \quad c = 0 < 1 = \log_b a, \]
  giving \( T(n) = \Theta(n^{\log_b a}) = \Theta(n). \)

- The runtime for **Merge-Sort** satisfies the recurrence:
  \[ T(n) = 2T(n/2) + \Theta(n). \]

  The Master Theorem (case 2) applies:
  \[ a = b = 2 \quad \text{and} \quad c = 1 = \log_b a, \]
  giving \( T(n) = \Theta(n^c \log n) = \Theta(n \log n). \)
What’s happening

For the recurrences

\[ T_1(n) = 4T(n/2) + n \]
\[ T_2(n) = 4T(n/2) + n^2 \]
\[ T_3(n) = 4T(n/2) + n^3 \]

the Master Theorem (case \( i \)) applies:

\[ a = 4 \quad \text{and} \quad b = 2 \quad (so \quad \log_b a = 2) \quad , \quad \text{and} \quad c = i \quad , \]

giving

\[ T_1(n) = \Theta(n^2) \quad , \quad T_2(n) = \Theta(n^2 \lg n) \quad , \quad \text{and} \quad T_3(n) = \Theta(n^3) \ . \]

Case 1: applies if the overhead cost \( (n^c) \) is negligible compared to the number and size of the subproblems.

Case 2: applies if the overhead cost \( (n^c) \) is as costly as the subproblems.

Case 3: applies if the overhead cost \( (n^c) \) is the dominating factor.

If we have Case 3, then in general this indicates, that the divide-and-conquer approach can be replaced by a simpler approach (as we have seen for the min-max algorithm).
Easy decision between the three cases

Consider (again)

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c). \]

The main question to start with is always:

**Which of the three cases applies?**

Apparently you needed to compute \( x = \log_b a \) for that. But it is actually easier:

1. If \( b^c < a \) then Case 1 applies.
2. If \( b^c = a \) then Case 2 applies.
3. If \( b^c > a \) then Case 3 applies.

(Try to understand why this holds — it’s easy.)
Further examples

- \( T(n) = 5T(n/2) + \Theta(n^2) \)
  
  In Master Theorem: \( a = 5, b = 2, c = 2 \).
  
  As \( \log_b a = \log_2 5 > \log_2 4 = 2 = c \), case 1 applies:
  
  \( T(n) = \Theta(n^{\log_2 5}) \).

- \( T(n) = 27T(n/3) + \Theta(n^3) \)
  
  In Master Theorem: \( a = 27, b = 3, c = 3 \).
  
  As \( \log_b a = \log_3 27 = 3 = c \), case 2 applies:
  
  \( T(n) = \Theta(n^3 \log n) \).

- \( T(n) = 5T(n/2) + \Theta(n^3) \)
  
  In Master Theorem: \( a = 5, b = 2, c = 3 \).
  
  As \( \log_b a = \log_2 5 < \log_2 8 = 3 = c \), case 3 applies:
  
  \( T(n) = \Theta(n^3) \).
A quick glance at matrix multiplication

Hopefully you recall from your first-year:

Multiplication of two $n \times n$ matrices is done by a three-times nested loop, and thus can be done in time $O(n^3)$.

For example

$$
\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{pmatrix}.
$$

Matrix multiplication is “everywhere” — can we do better than $O(n^3)$?
First approach

The previous formula for multiplication of $2 \times 2$ matrices can be used for arbitrary matrix multiplication for $n \times n$ matrices:

- Subdivide each of the two matrices into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$.
- Apply the formula, plugging in the sub-matrices.

We have 8 multiplications with the smaller matrices, yielding

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2).$$

which yields

$$T(n) = \Theta(n^3).$$

We needed to do better ?!
First approach

The previous formula for multiplication of $2 \times 2$ matrices can be used for arbitrary matrix multiplication for $n \times n$ matrices:

- Subdivide each of the two matrices into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$.
- Apply the formula, plugging in the sub-matrices.

We have 8 multiplications with the smaller matrices, yielding

$$T(n) = 8 T\left(\frac{n}{2}\right) + \Theta(n^2).$$

which yields

$$T(n) = \Theta(n^3).$$

We needed to do better ?!
A trick with astounding effects

The trick is now to do the multiplication of two $2 \times 2$ matrices with only 7 multiplications:

1. We need more additions, but this is irrelevant here.
2. The savings comes from using intermediate results (factoring out ...).
3. See the tutorial for how this is done.

We then get the recurrence

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2).$$

We have $\lg 7 \approx 2.807355$, and thus

$$T(n) = \Theta(n^{2.807356}).$$
Remarks on Merge-Sort

Stability

For many sorting-applications, the objects to be sorted consist of a **key** which provides the sorting criterion, and a lot of other data; for example the last name as part of an employee-record. Then it is quite natural that different objects have the same key. Often, such arrays are then pre-sorted according to other criterions.

“Stability” of a sorting algorithm is now the property that the order of equal elements (according to their keys) is not changed. Merge-Sort is stable (at least in our implementation; also Insertion-Sort is stable).
Remarks on Merge-Sort (cont.)

In-place  A sorting algorithms sorts “in-place” if besides the given array and some auxiliary data is doesn’t need more memory. This is important if the array is very large (say, $n \approx 10^9$).
Insertion-Sort is in-place, while our algorithm for Merge-Sort is not (needing $\approx 2n$ memory cells).
One can make Merge-Sort in-place, but this (apparently) only with a complicated algorithm, which in practice seems not to be applied. If in-place sorting is required, then often one uses “Heap-Sort”.

Already sorted  If the array is already sorted, then only $n - 1$ comparisons are needed (however overall it still needs time $\Theta(n \log n)$ because of the swapping, and it stills needs space $2n$).
Remarks on Merge-Sort (cont.)

**Overhead** The general overhead of Merge-Sort (due to the swapping) is somewhat higher than what can be achieved with “Quick-Sort”, which typically is the default sorting-algorithm in libraries.
The minimal numbers of comparisons

Let $S(n)$ be the minimum number of comparisons that will (always!) suffice to sort $n$ elements (using only comparisons between the elements, and no other properties of them). It holds

$$S(N) \geq \lceil \lg(n!) \rceil = \Theta(n \log n).$$

This is the so-called information-theoretic lower bound: It follows by observing that the $n!$ many ordering of $1, \ldots, n$ need to be handled, where every comparison establishes 1 bit of information.

The initial known precise values for $S(n)$ are:

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(n)$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>34</td>
<td>38</td>
</tr>
</tbody>
</table>

The first open value is $S(15)$ (see http://oeis.org/A036604).
More recurrences

1. \( T(n) = 3T(n/3) + 1 : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 \): \( T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 \): \( T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 \):
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : \)
More recurrences

1. $T(n) = 3T(n/3) + 1 : T(n) = \Theta(n)$
2. $T(n) = aT(n/a) + 1 : T(n) = \Theta(n)$
3. $T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2})$
4. $T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4})$
5. $T(n) = 3T(n/3) + n :$
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : T(n) = \Theta(n^{\log_3 4} \log n) \)
9. \( T(n) = 3T(n/3) + n^{1.5} : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : T(n) = \Theta(n^{\log_3 4} \log n) \)
9. \( T(n) = 3T(n/3) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
10. \( T(n) = aT(n/a) + n^{1.5} : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : T(n) = \Theta(n^{\log_3 4} \log n) \)
9. \( T(n) = 3T(n/3) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
10. \( T(n) = aT(n/a) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
11. \( T(n) = 2T(n/3) + n : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)

5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : T(n) = \Theta(n^{\log_3 4} \log n) \)

9. \( T(n) = 3T(n/3) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
10. \( T(n) = aT(n/a) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
11. \( T(n) = 2T(n/3) + n : T(n) = \Theta(n) \)
12. \( T(n) = 4T(n/3) + n^2 : \)
More recurrences

1. \( T(n) = 3T(n/3) + 1 : T(n) = \Theta(n) \)
2. \( T(n) = aT(n/a) + 1 : T(n) = \Theta(n) \)
3. \( T(n) = 2T(n/3) + 1 : T(n) = \Theta(n^{\log_3 2}) \)
4. \( T(n) = 4T(n/3) + 1 : T(n) = \Theta(n^{\log_3 4}) \)
5. \( T(n) = 3T(n/3) + n : T(n) = \Theta(n \log n) \)
6. \( T(n) = aT(n/a) + n : T(n) = \Theta(n \log n) \)
7. \( T(n) = 2T(n/3) + n^{\log_3 2} : T(n) = \Theta(n^{\log_3 2} \log n) \)
8. \( T(n) = 4T(n/3) + n^{\log_3 4} : T(n) = \Theta(n^{\log_3 4} \log n) \)
9. \( T(n) = 3T(n/3) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
10. \( T(n) = aT(n/a) + n^{1.5} : T(n) = \Theta(n^{1.5}) \)
11. \( T(n) = 2T(n/3) + n : T(n) = \Theta(n) \)
12. \( T(n) = 4T(n/3) + n^2 : T(n) = \Theta(n^2) \)
How to do it with 7 multiplications

\[
\begin{pmatrix}
a_{1,1} & a_{1,2} \\
a_{2,1} & a_{2,2}
\end{pmatrix}
\cdot
\begin{pmatrix}
b_{1,1} & b_{1,2} \\
b_{2,1} & b_{2,2}
\end{pmatrix}
= \\
\begin{pmatrix}
a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\
a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2}
\end{pmatrix}
\]

Create 10 auxiliary results:

\[
\begin{align*}
S_1 &= b_{1,2} - b_{2,2} \\
S_2 &= a_{1,1} + a_{1,2} \\
S_3 &= a_{2,1} + a_{2,2} \\
S_4 &= b_{2,1} - b_{1,1} \\
S_5 &= a_{1,1} + a_{2,2} \\
S_6 &= b_{1,1} + b_{2,2} \\
S_7 &= a_{1,2} - a_{2,2} \\
S_8 &= b_{2,1} + b_{2,2} \\
S_9 &= a_{1,1} - a_{2,1} \\
S_{10} &= b_{1,1} + b_{2,1}
\end{align*}
\]
How to do it with 7 multiplications (cont.)

Perform 7 multiplications:

\[
\begin{align*}
P_1 &= a_{1,1} \cdot S_1 = a_{1,1} \cdot b_{1,2} - a_{1,1} \cdot b_{2,2} \\
P_2 &= S_2 \cdot b_{2,2} = a_{1,1} \cdot b_{2,2} + a_{1,2} \cdot b_{2,2} \\
P_3 &= S_3 \cdot b_{1,1} = a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{1,1} \\
P_4 &= a_{2,2} \cdot S_4 = a_{2,2} \cdot b_{2,1} - a_{2,2} \cdot b_{1,1} \\
P_5 &= S_5 \cdot S_6 = a_{1,1} \cdot b_{1,1} + a_{1,1} \cdot b_{2,2} + a_{2,2} \cdot b_{1,1} + a_{2,2} \cdot b_{2,2} \\
P_6 &= S_7 \cdot S_8 = a_{1,2} \cdot b_{2,1} + a_{1,2} \cdot b_{2,2} - a_{2,2} \cdot b_{2,1} - a_{2,2} \cdot b_{2,2} \\
P_7 &= S_9 \cdot S_{10} = a_{1,1} \cdot b_{1,1} + a_{1,1} \cdot b_{1,2} - a_{2,1} \cdot b_{1,1} - a_{2,1} \cdot b_{1,2}
\end{align*}
\]

Harvest:

\[
\begin{align*}
a_{1,1}b_{1,1} + a_{1,2}b_{2,1} &= P_5 + P_4 - P_2 + P_6 \\
a_{1,1}b_{1,2} + a_{1,2}b_{2,2} &= P_1 + P_2 \\
a_{2,1}b_{1,1} + a_{2,2}b_{2,1} &= P_3 + P_4 \\
a_{2,1}b_{1,2} + a_{2,2}b_{2,2} &= P_5 + P_1 - P_3 - P_7.
\end{align*}
\]