The Master Method

Advanced Algorithms and Data Structures - Lecture 2A

Venanzio Capretta
Thursday 8 October 2020

School of Computer Science, University of Nottingham
Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

\[ l = [4, -2, 3, -7, 5, 2, -3, 4, -8, 6, -2, 1] \]
Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

\[ l_1 = [4, -2, 3, -7, 5, 2] \mid [-3, 4, -8, 6, -2, 1] = l_2 \]

- **Split** the input array in two halves
Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

\[ l_1 = [4, -2, 3, -7, 5, 2] ] [ -3, 4, -8, 6, -2, 1] = l_2 \]

- **Split** the input array in two halves
- **Compute the maximum subarray of each half**
Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

\[ l_1 = [4, -2, 3, -7, 5, 2, \overbrace{-3, 4, -8, 6}^\text{maxSub } l_2] = l_2 \]

- **Split** the input array in two halves
- **Compute the maximum subarray of each half**
- **Compute the maximum cross-over subarray**
Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

\[ l_1 = [4, -2, 3, -7, \underbrace{5, 2}_\text{maxSub } l_1] \cdot [ -3, 4, -8, \underbrace{6}_\text{maxSub } l_2, -2, 1] = l_2 \]

- **Split** the input array in two halves
- **Compute the maximum subarray of each half**
- **Compute the maximum cross-over subarray**

The result is the maximum of the three partial subproblems
Maximum Array DC in Haskell

maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = let mid = length xs ‘div‘ 2
          (xs1,xs2) = splitAt mid xs
          (i1,j1,max1) = maxSub xs1
          (i2,j2,max2) = maxSub xs2
          (i3,j3,max3) = maxCross xs1 xs2
          in if max1 ≥ max2 && max1 ≥ max3 then (i1,j1,max1)
             else if max2 ≥ max3 then (i2+mid,j2+mid,max2)
             else (i3,j3+mid,max3)

maxCross is an auxiliary functions that finds the maximum crossover sublist, with i3 the start index in xs1 and j3 the end index in xs2.

It has linear complexity in the sum of the lengths of xs1 and xs2.
Let’s determine the time complexity $T(n)$ of this algorithm.
Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$
Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:
Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
Recursive Equations for Time Complexity

Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function $\text{maxCross}$ has linear time complexity, $c_2 n$
Recursive Equations for Time Complexity

Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function $\text{maxCross}$ has linear time complexity, $c_2 n$
- Determining the largest among the three partial results $\text{max1}$, $\text{max2}$, and $\text{max3}$ and returning the corresponding output takes constant time $d$
Recursive Equations for Time Complexity

Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function $\text{maxCross}$ has linear time complexity, $c_2 n$
- Determining the largest among the three partial results $\text{max1}$, $\text{max2}$, and $\text{max3}$ and returning the corresponding output takes constant time $d$
- Finally the two recursive calls $\text{maxSub } xs1$ and $\text{maxSub } xs2$ will each take time $T(n/2)$ because $xs1$ and $xs1$ have half the size of $xs$
Recursive Equations for Time Complexity

Let’s determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function $\text{maxCross}$ has linear time complexity, $c_2 n$
- Determining the largest among the three partial results $\text{max1}$, $\text{max2}$, and $\text{max3}$ and returning the corresponding output takes constant time $d$
- Finally the two recursive calls $\text{maxSub}$ $\text{xs1}$ and $\text{maxSub}$ $\text{xs2}$ will each take time $T(n/2)$ because $\text{xs1}$ and $\text{xs1}$ have half the size of $\text{xs}$

Putting all the components together we get (with $c = c_1 + c_2$):

$$T(n) = 2T(n/2) + c_1 n + c_2 n + d = 2T(n/2) + cn + d$$
Strictly speaking, if the length $n$ of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$. The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class.
Simplifying the Equations

Strictly speaking, if the length $n$ of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$

The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class

The exact value of the constant factors is not relevant

The larger component dominates

So in $cn + d$ we just need to consider that this term is linear
Simplifying the Equations

Strictly speaking, if the length $n$ of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$

The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class.

The exact value of the constant factors is not relevant.

The larger component dominates.

So in $cn + d$ we just need to consider that this term is linear.

We can rewrite the equation using complexity classes for the terms:

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n/2) + \Theta(n)$$
Solving Recursive Equations

Three methods to solve a recursive equation:

- **Substitution Method**: make a guess on the complexity class, verify and derive the parameters by recursion
- **Recursion Tree Method**: Draw a tree with all the recursive calls of the function and add up all the steps in each node
- **Master Method**: A general theorem that gives you the complexity class depending on the form of the equation
Solving Recursive Equations

Three methods to solve a recursive equation:

- **Substitution Method**: make a guess on the complexity class, verify and derive the parameters by recursion
- **Recursion Tree Method**: Draw a tree with all the recursive calls of the function and add up all the steps in each node
- **Master Method**: A general theorem that gives you the complexity class depending on the form of the equation

Let’s apply all three to the simplified system of equations

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= 2T(n/2) + n
\end{align*}
\]

The solution will be the same as for the equations for the Maximum Subarray algorithm (and merge sort)
Guess the solution:
Since it is the same equation as for merge sort, we guess that

\[ T(n) = O(n \log n) \]
Guess the solution:
Since it is the same equation as for merge sort, we guess that

\[ T(n) = O(n \log n) \]

By the definition of \( O \)-notation, this means that
There exists a factor \( c \) and a starting size \( n_0 \) such that:

\[ T(n) \leq cn \log n \quad \text{for } n \geq n_0 \]
Guess the solution:
Since it is the same equation as for merge sort, we guess that

\[ T(n) = O(n \log n) \]

By the definition of \( O \)-notation, this means that

There exists a factor \( c \) and a starting size \( n_0 \) such that:

\[ T(n) \leq cn \log n \quad \text{for } n \geq n_0 \]

Let’s check that this works for the inductive step:
Assume that it is true for values smaller than \( n \)
Prove that it also must hold for \( n \):

\[
T(n) = 2T(n/2) + n \\
\leq 2c \frac{n}{2} \log \frac{n}{2} + n \quad \text{by Induction Hypothesis} \\
= cn(\log n - \log 2) + n = cn(\log n - 1) + n \\
= cn \log n - cn + n \leq cn \log n \quad \text{if } c \geq 1
\]
The base case is more problematic:
We have $T(1) = 1$, we can’t prove $T(1) \leq c_1 \log 1 = 0$
The base case is more problematic:

We have \( T(1) = 1 \), we can’t prove \( T(1) \leq c_1 \log 1 = 0 \)

But we can choose any starting point \( n_0 \)

For example

\[
T(2) = 2T(1) + 2 = 4 \\
\leq c_2 \log 2 = 2c \quad \text{if } c \geq 2
\]
The base case is more problematic:
We have \( T(1) = 1 \), we can’t prove \( T(1) \leq c_1 \log 1 = 0 \)

But we can choose any starting point \( n_0 \)
For example

\[
T(2) = 2T(1) + 2 = 4 \\
\leq c_2 \log 2 = 2c \quad \text{if } c \geq 2
\]

So everything works if we choose \( n_0 = 2 \) and \( c = 2 \)

We proved that \( T(n) = O(n \log n) \)

(We’ve been a bit simplistic: \( n/2 \) is not guaranteed to be an integer. Either assume that \( n \) is a power of two, or replace \( n/2 \) with \( \lfloor n/2 \rfloor \))
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments:
Root: $T(n)$  
Children: two calls $T(n/2)$  
And so on
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments
Root: $T(n)$  Children: two calls $T(n/2)$  And so on

\[
\begin{align*}
n & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{Root:} \quad T(n) \quad \text{Children: two calls} \quad T(n/2) \quad \text{And so on}
\end{align*}
\]
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: \( T(n) \) \hspace{1cm} Children: two calls \( T(n/2) \) \hspace{1cm} And so on

\[
\begin{align*}
\text{Root: } & \quad n \\
\text{Children: } & \quad n/2, n/2 \\
& \quad n/4, n/4, n/4, n/4 \\
& \quad n/8, n/8, n/8, n/8, n/8, n/8, n/8, n/8 \\
& \quad \vdots \\
& \quad n/2^k = 1, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, 1 \\
\end{align*}
\]

What is the depth \( k \)?
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$  
Children: two calls $T(n/2)$  
And so on

What is the depth $k$?  
$k = \log n$
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$  
Children: two calls $T(n/2)$  
And so on

What is the depth $k$?  $k = \log n$

How many computation steps do we do at each node?
Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$  Children: two calls $T(n/2)$  And so on

What is the depth $k$?  $k = \log n$

How many computation steps do we do at each node?  At level $j$, $n/2^j$
Let’s sum up all the computation steps:
Let’s sum up all the computation steps:

- There are $k = \log n$ levels in the tree
Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level $j$ there are $2^j$ nodes with argument $n/2^j$
Let’s sum up all the computation steps:

- There are \( k = \log n \) levels in the tree
- At each level \( j \) there are \( 2^j \) nodes with argument \( n/2^j \)
- The recursive equation for those nodes gives

\[
T(n/2^j) = 2T(n/2^{j+1}) + n/2^j
\]

So the computation steps for each node is \( n/2^j \)
Let’s sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level $j$ there are $2^j$ nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2T(n/2^{j+1}) + n/2^j$$

So the computation steps for each node is $n/2^j$
- Adding up all the steps at level $j$ we get: $2^j n/2^j = n$
Let’s sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level $j$ there are $2^j$ nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2T(n/2^{j+1}) + n/2^j$$

So the computation steps for each node is $n/2^j$
- Adding up all the steps at level $j$ we get: $2^j n/2^j = n$

So there are a total of $n$ computation steps at each level and there are $\log n$ levels

**Total number of steps:** $n \log n$
Recursion Tree Method, Calculation

Let’s sum up all the computation steps:

- There are $k = \log n$ levels in the tree.
- At each level $j$ there are $2^j$ nodes with argument $n/2^j$.
- The recursive equation for those nodes gives
  \[ T(n/2^j) = 2T(n/2^{j+1}) + n/2^j \]
  So the computation steps for each node is $n/2^j$.
- Adding up all the steps at level $j$ we get: $2^j n/2^j = n$.

So there are a total of $n$ computation steps at each level and there are $\log n$ levels.

**Total number of steps:** $n \log n$.

This shows that $T(n) = \Theta(n \log n)$. 
The **Master Method** generalizes the recursion tree techniques to algorithms with different number of recursive calls with different sizes of arguments.
The Master Method generalizes the recursion tree techniques to algorithms with different number of recursive calls with different sizes of arguments.

The Maximum Subarray algorithm (and Merge Sort) had:

- Two recursive calls
- Each with an argument of half size, \( n/2 \)
- A linear non-recursive part

This leads to the equation: \( T(n) = 2T(n/2) + cn \)
The Master Method generalizes the recursion tree techniques to algorithms with different number of recursive calls with different sizes of arguments.

The Maximum Subarray algorithm (and Merge Sort) had:

- Two recursive calls
- Each with an argument of half size, $n/2$
- A linear non-recursive part

This leads to the equation: $T(n) = 2T(n/2) + cn$

A more general recursive program could have:

- Any number ($a$) of recursive calls
- Each with an argument of size $n/b$
- A non-recursive part given by a function $f(n)$

This leads to the equation $T(n) = aT(n/b) + f(n)$
If we draw the recursion tree:
If we draw the recursion tree:

- Number of children for each node: $a$
If we draw the recursion tree:

- Number of children for each node: $a$
- Arguments at level $j$: 

---

Master Method: Recursion Trees
If we draw the recursion tree:

- Number of children for each node: \( a \)
- Arguments at level \( j \): \( n/b^j \)
If we draw the recursion tree:

- Number of children for each node: $a$
- Arguments at level $j$: $n/b^j$
- Depth of tree:
If we draw the recursion tree:

- Number of children for each node: $a$
- Arguments at level $j$: $n/b^j$
- Depth of tree: $\log_b n$
If we draw the recursion tree:

- Number of children for each node: $a$
- Arguments at level $j$: $n/b^j$
- Depth of tree: $\log_b n$

What is the total number of nodes?
Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: \( a \)
- Arguments at level \( j \): \( n/b^j \)
- Depth of tree: \( \log_b n \)

What is the total number of nodes?

- 1 node at level 0 (root)
If we draw the recursion tree:

- Number of children for each node: \( a \)
- Arguments at level \( j \): \( n/b^j \)
- Depth of tree: \( \log_b n \)

What is the total number of nodes?

- 1 node at level 0 (root)
- \( a \) nodes at level 1
If we draw the recursion tree:

- Number of children for each node: \( a \)
- Arguments at level \( j \): \( \frac{n}{b^j} \)
- Depth of tree: \( \log_b n \)

What is the total number of nodes?

- 1 node at level 0 (root)
- \( a \) nodes at level 1
- \( a^2 \) nodes at level 2
If we draw the recursion tree:

- Number of children for each node: \( a \)
- Arguments at level \( j \): \( \frac{n}{b^j} \)
- Depth of tree: \( \log_b n \)

What is the total number of nodes?

- 1 node at level 0 (root)
- \( a \) nodes at level 1
- \( a^2 \) nodes at level 2
- \( a^j \) nodes at level \( j \)

There are \( k = \log_b n \) levels, total number of nodes:

\[
1 + a + a^2 + a^3 + \cdots + a^{\log_b n}
\]

This is a geometric series (see IA Appendix A)
Total number of nodes:

\[ \sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} \]
Master Method: Computation Steps

Total number of nodes:

\[ \sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n}) \]
Master Method: Computation Steps

Total number of nodes:

\[
\sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})
\]

Compare with the non-recursive part \( f(n) \):
Master Method: Computation Steps

Total number of nodes:

\[
\sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})
\]

Compare with the non-recursive part \( f(n) \):

- If the non-recursive part grows slower than the number of nodes:
  \[
  f(n) = O(n^{\log_b a - \epsilon}) \quad \text{for some } \epsilon > 0
  \]
  the recursive part dominates: \( T(n) = \Theta(n^{\log_b a}) \)
Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$$

Compare with the non-recursive part $f(n)$:

- If the non-recursive part grows slower than the number of nodes:
  $$f(n) = O(n^{\log_b a - \epsilon}) \quad \text{for some } \epsilon > 0$$
  the recursive part dominates: $$T(n) = \Theta(n^{\log_b a})$$

- If they are of the same class: $$f(n) = \Theta(n^{\log_b a})$$
  each level adds $n^{\log_b a}$ computation steps (check the math)
  There are $\log_b n$ levels, so: $$T(n) = \Theta(n^{\log_b a \log_b n}) = \Theta(n^{\log_b a \log n})$$
Master Method: Computation Steps

Total number of nodes:

\[ \sum_{j=0}^{j=k} a^j = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \]

Compare with the non-recursive part \( f(n) \):

- If the non-recursive part grows slower than the number of nodes:
  \[ f(n) = O(n^{\log_b a - \epsilon}) \text{ for some } \epsilon > 0 \]
  the recursive part dominates: \( T(n) = \Theta(n^{\log_b a}) \)

- If they are of the same class: \( f(n) = \Theta(n^{\log_b a}) \)
  each level adds \( n^{\log_b a} \) computation steps (check the math)
  There are \( \log_b n \) levels, so: \( T(n) = \Theta(n^{\log_b a \log_b n}) = \Theta(n^{\log_b a \log n}) \)

- If the non-recursive part grows faster than the number of nodes:
  \[ f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ for some } \epsilon > 0 \]
  (plus some other condition)
  the non-recursive part dominates: \( T(n) = \Theta(f(n)) \)
In the case of the Maximum Array algorithm (and Merge Sort):

\[ T(1) = c_0 \]
\[ T(n) = 2T(n/2) + c_1 n + c_2 \]
In the case of the Maximum Array algorithm (and Merge Sort):

\[ T(1) = c_0 \]
\[ T(n) = 2T(n/2) + c_1 n + c_2 \]

We have \( a = 2, \ b = 2, \ f(n) = c_1 n + c_2 \)
In the case of the Maximum Array algorithm (and Merge Sort):

\[
T(1) = c_0 \\
T(n) = 2T(n/2) + c_1 n + c_2
\]

We have \( a = 2, \ b = 2, \ f(n) = c_1 n + c_2 \)

We must compare \( f(n) \) with \( n^{\log_b a} = n^{\log_2 2} = n \)

We have \( f(n) = \Theta(n) \), so we’re in the second case
In the case of the Maximum Array algorithm (and Merge Sort):

\[
T(1) = c_0 \\
T(n) = 2T(n/2) + c_1 n + c_2
\]

We have \( a = 2, \ b = 2, \ f(n) = c_1 n + c_2 \)

We must compare \( f(n) \) with \( n^{\log_b a} = n^{\log_2 2} = n \)

We have \( f(n) = \Theta(n) \), so we’re in the second case

Conclusion \( T(n) = \Theta(n^{\log_b a \log n}) = \Theta(n \log n) \)