Algorithms and Datastructures

Divide and Conquer, Master theorem

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Datastructures, March 2018

Structure

Concept:

Divide and Conquer

Introduction

Concept:

- Divide the problem into smaller subproblems

Divide and Conquer
Introduction

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly
- Connect all subsolutions to solve the overall problem

Divide and Conquer
Introduction

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly
- Connect all subsolutions to solve the overall problem
- Recursive application of the algorithm on smaller subproblems

Divide and Conquer

Introduction

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly
- Connect all subsolutions to solve the overall problem
- Recursive application of the algorithm on smaller subproblems
- Direct solving of small subproblems

Structure

Input:

Output:

Divide and Conquer

Maximum Subtotal

Input:

- Sequence X of n integers

Output:

Input:

- Sequence X of n integers

Output:

- Maximum sum of an uninterrupted subsequence of X and its index boundary

Input:

- Sequence X of n integers

Output:

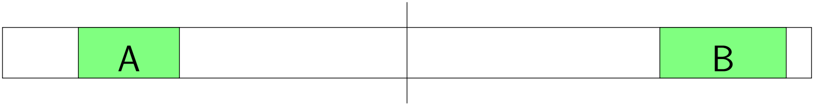
- Maximum sum of an uninterrupted subsequence of X and its index boundary

Table: Input values

Index	0	1	2	3	4	5	6	7	8	9
Value	31	-41	59	26	-53	58	97	-93	-23	84

Output: Sum: 187, Start: 2, End: 6

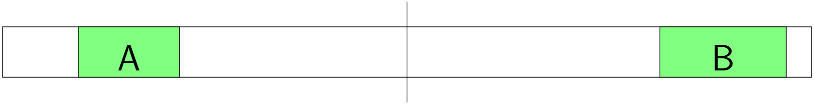
Idea:



Divide and Conquer

Maximum Subtotal

Idea:

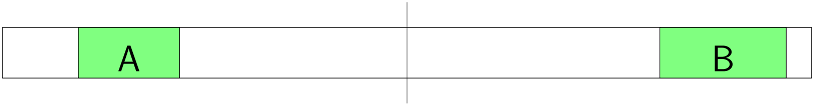


- Solve the left / right half of the problem recursive

Divide and Conquer

Maximum Subtotal

Idea:

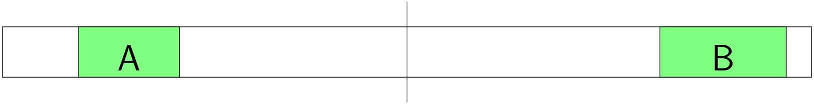


- Solve the left / right half of the problem recursive
- Combine both solutions into a overall solution

Divide and Conquer

Maximum Subtotal

Idea:

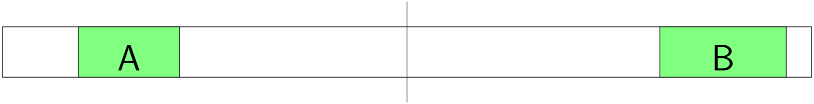


- Solve the left / right half of the problem recursive
- Combine both solutions into a overall solution
- The maximum is located in the left half (A) or the right half (B)

Divide and Conquer

Maximum Subtotal

Idea:

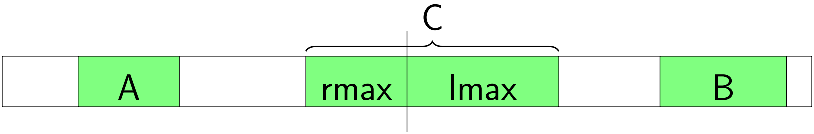


- Solve the left / right half of the problem recursive
- Combine both solutions into a overall solution
- The maximum is located in the left half (A) or the right half (B)
- The maximum interval can overlap with the border (C)

Divide and Conquer

Maximum Subtotal

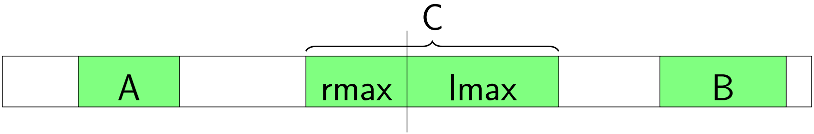
Principle:



Divide and Conquer

Maximum Subtotal

Principle:

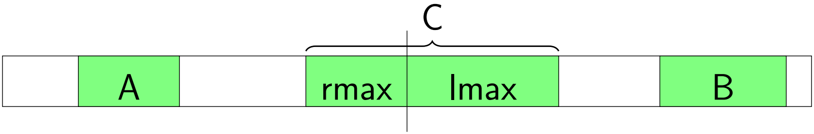


- Small problems are solved directly: $n = 1 \Rightarrow \text{max} = X[0]$

Divide and Conquer

Maximum Subtotal

Principle:

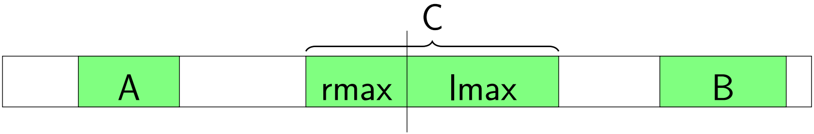


- Small problems are solved directly: $n = 1 \Rightarrow \max = X[0]$
- Big problems are decomposed into two subproblems and solved recursively. Subsolutions A and B are returned

Divide and Conquer

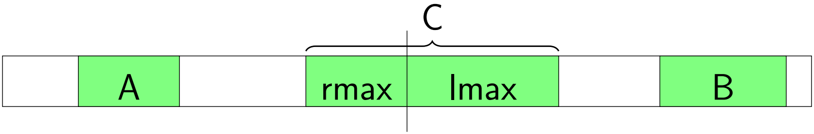
Maximum Subtotal

Principle:



- Small problems are solved directly: $n = 1 \Rightarrow \text{max} = X[0]$
- Big problems are decomposed into two subproblems and solved recursively. Subsolutions A and B are returned
- To solve C we have to calculate rmax and lmax

Principle:



- Small problems are solved directly: $n = 1 \Rightarrow \max = X[0]$
- Big problems are decomposed into two subproblems and solved recursively. Subsolutions A and B are returned
- To solve C we have to calculate rmax and lmax
- Overall solution is maximum of A B and C

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) / 2
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)
```

Divide and Conquer

Maximum Subtotal - Python

```python
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    # rmax and lmax for cornercase C
    C1, C2 = rmax(X, i, m), lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    # rmax and lmax for cornercase C
    C1, C2 = rmax(X, i, m), lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    # compute solution from results A, B, C
    return max([A, B, C], key=lambda i: i[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
#Alternative trivial case
def maxSubArray(X, i, j):
```

Divide and Conquer

Maximum Subtotal - Python

```
#Alternative trivial case
def maxSubArray(X, i, j):
    # trivial: only one element
    if i == j:
        return (X[i], i, i)
```

Divide and Conquer

Maximum Subtotal - Python

```
#Alternative trivial case
def maxSubArray(X, i, j):
    # trivial: only one element
    if i == j:
        return (X[i], i, i)

    # trivial: only two elements
    if i + 1 == j:
        return max([
            (X[i], i, i),
            (X[j], j, j),
            (X[i] + X[j], i, j)
        ], key=lambda item: item[0])

    ... # continue as before
```

Divide and Conquer

Maximum Subtotal - Python

```python
#Implementation max
def max(a, b, c):
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation max
def max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
```

Divide and Conquer

Maximum Subtotal - Python

```python
#Implementation max
def max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if c > b:
            return c
        else:
            return b
```

Divide and Conquer

Maximum Subtotal - Python

```
#Alternative implementation max
```

Divide and Conquer

Maximum Subtotal - Python

```python
#Alternative implementation max

def max(a, b):
    if a > b:
        return a
    else:
        return b
```

Divide and Conquer

Maximum Subtotal - Python

```python
#Alternative implementation max

def max(a, b):
    if a > b:
        return a
    else:
        return b

def maxTripel(a, b, c):
    return max(max(a,b),c)
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation left maximum
def lmax(X, i, j):
    maxSum = (X[i], i)
    s = X[i]

    # sum up from the lower index going up
    # (from left to right)
    for k in range(i+1, j+1):
        s += X[k]

        if s > maxSum[0]:
            maxSum = (s, k)

    return maxSum
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation right maximum
def rmax(X, i, j):
    maxSum = (X[j], j)
    s = X[i]

    # sum up from the upper index going down
    # (from right to left)
    for k in range(j-1, i-1, -1):
        s += X[k]

        if s > maxSum[0]:
            maxSum = (s, k)

    return maxSum
```

Divide and Conquer

Maximum Subtotal

Table: *lmax* example

index	i	$i+1$	$\cdots$	$\cdots$	$j-1$	j
X	58	-53	26	59	-41	31
sum	58	5	31	90	49	80
lmax	58	58	58	90	90	90

Divide and Conquer

Maximum Subtotal

Table: *lmax* example

index	i	$i+1$	$\cdots$	$\cdots$	$j-1$	j
X	58	-53	26	59	-41	31
sum	58	5	31	90	49	80
lmax	58	58	58	90	90	90

- The *sum* and *lmax* are initialized with $X[i]$

Divide and Conquer

Maximum Subtotal

Table: *lmax* example

index	i	$i+1$	$\cdots$	$\cdots$	$j-1$	j
X	58	-53	26	59	-41	31
sum	58	5	31	90	49	80
lmax	58	58	58	90	90	90

- The *sum* and *lmax* are initialized with $X[i]$
- We iterate over X from $i+1$ to j and update *sum*

Divide and Conquer

Maximum Subtotal

Table: *lmax* example

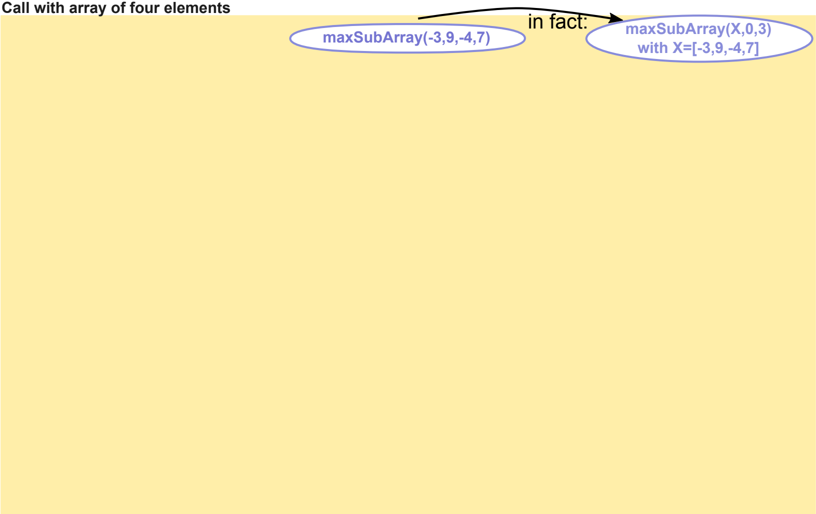
index	i	$i+1$	$\cdots$	$\cdots$	$j-1$	j
X	58	-53	26	59	-41	31
sum	58	5	31	90	49	80
lmax	58	58	58	90	90	90

- The *sum* and *lmax* are initialized with $X[i]$
- We iterate over X from $i+1$ to j and update *sum*
- If $sum > lmax$ then *lmax* gets updated

Divide and Conquer

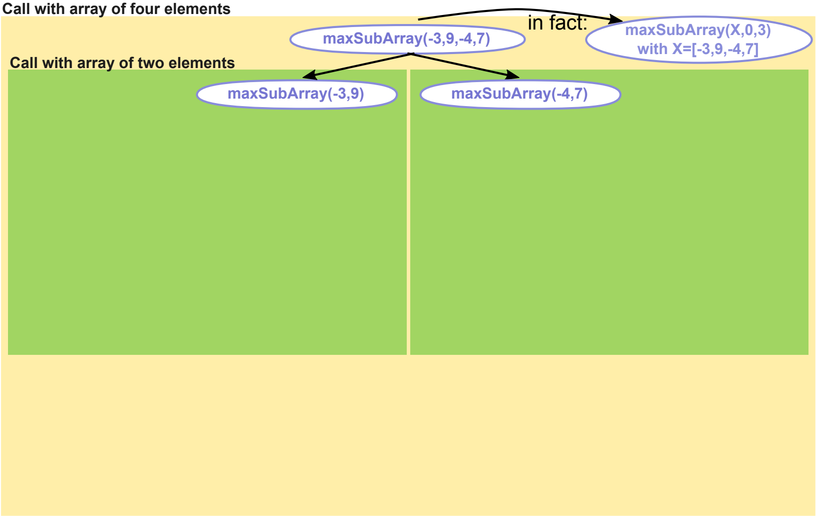
Maximum Subtotal

Call with array of four elements



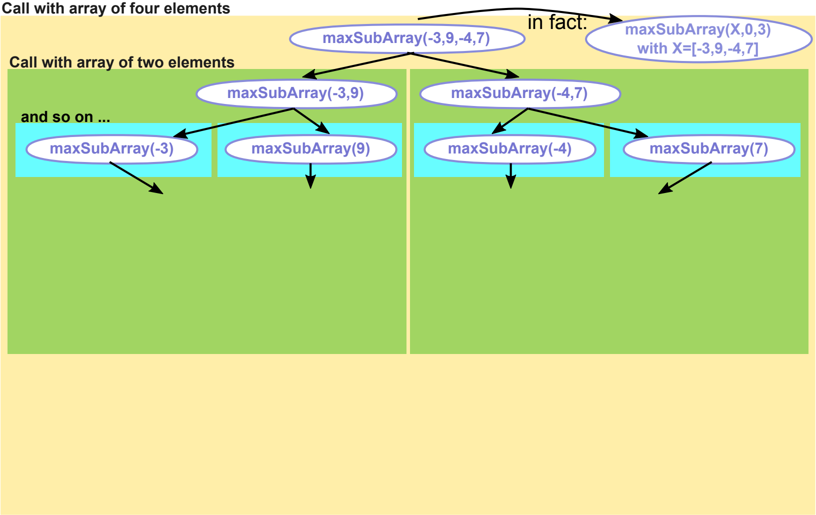
Divide and Conquer

Maximum Subtotal



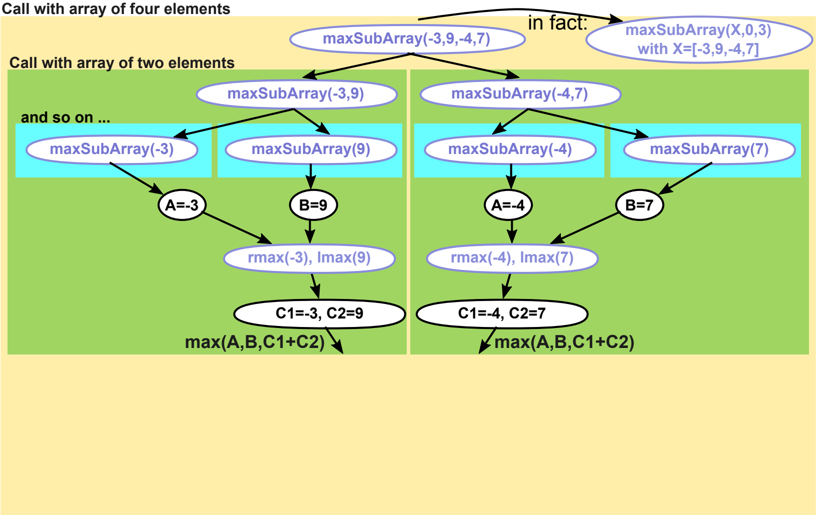
Divide and Conquer

Maximum Subtotal



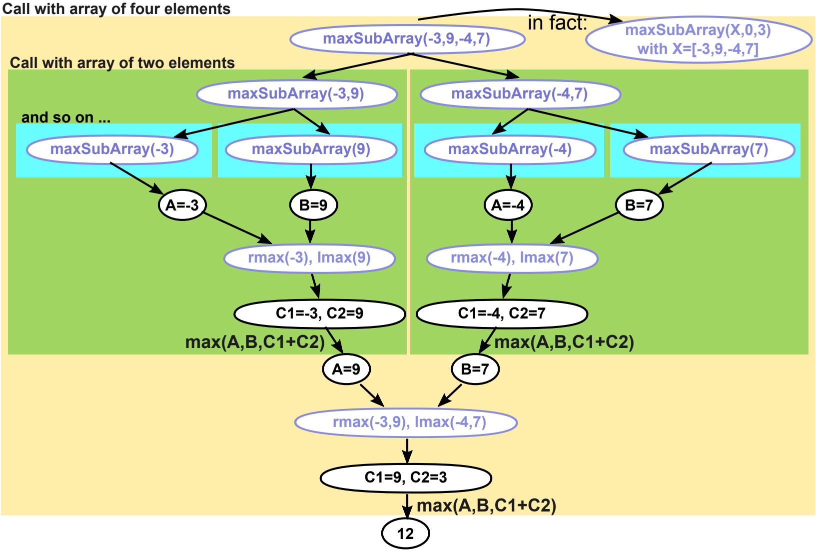
Divide and Conquer

Maximum Subtotal



Divide and Conquer

Maximum Subtotal



Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:
        return (X[i], i, i)

    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```python
def maxSubArray(X, i, j):
    if i == j:
        return (X[i], i, i)

    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                    # O(1)
        return (X[i], i, i)

    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                          # O(1)
        return (X[i], i, i)                             # O(1)

    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```python
def maxSubArray(X, i, j):
    if i == j:                                   # O(1)
        return (X[i], i, i)                      # O(1)

    m = (i + j) / 2                              # O(1)
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                  # O(1)
        return (X[i], i, i)                     # O(1)

    m = (i + j) / 2                             # O(1)
    A = maxSubArray(X, i, m)                    # T(n/2)
    B = maxSubArray(X, m + 1, j)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                  # O(1)
        return (X[i], i, i)                     # O(1)

    m = (i + j) / 2                             # O(1)
    A = maxSubArray(X, i, m)                    # T(n/2)
    B = maxSubArray(X, m + 1, j)                # T(n/2)

    C1 = rmax(X, i, m)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```python
def maxSubArray(X, i, j):
    if i == j:                                          # O(1)
        return (X[i], i, i)                             # O(1)

    m = (i + j) / 2                                     # O(1)
    A = maxSubArray(X, i, m)                            # T(n/2)
    B = maxSubArray(X, m + 1, j)                        # T(n/2)

    C1 = rmax(X, i, m)                                  # O(n)
    C2 = lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                   # O(1)
        return (X[i], i, i)                      # O(1)

    m = (i + j) / 2                              # O(1)
    A = maxSubArray(X, i, m)                     # T(n/2)
    B = maxSubArray(X, m + 1, j)                 # T(n/2)

    C1 = rmax(X, i, m)                           # O(n)
    C2 = lmax(X, m + 1, j)                       # O(n)
    C = (C1[0] + C2[0], C1[1], C2[1])

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                   # O(1)
        return (X[i], i, i)                      # O(1)

    m = (i + j) / 2                              # O(1)
    A = maxSubArray(X, i, m)                     # T(n/2)
    B = maxSubArray(X, m + 1, j)                 # T(n/2)

    C1 = rmax(X, i, m)                           # O(n)
    C2 = lmax(X, m + 1, j)                       # O(n)
    C = (C1[0] + C2[0], C1[1], C2[1])            # O(1)

    return max([A, B, C], \
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j:                                  # O(1)
        return (X[i], i, i)                     # O(1)

    m = (i + j) / 2                             # O(1)
    A = maxSubArray(X, i, m)                    # T(n/2)
    B = maxSubArray(X, m + 1, j)                # T(n/2)

    C1 = rmax(X, i, m)                          # O(n)
    C2 = lmax(X, m + 1, j)                      # O(n)
    C = (C1[0] + C2[0], C1[1], C2[1])           # O(1)

    return max([A, B, C], \                     # O(1)
        key=lambda item: item[0])
```

Divide and Conquer

Maximum Subtotal - Number of steps $T(n)$

Recursion equation:

$$T(n) = \begin{cases} \underbrace{\Theta(1)}_{\text{trivial case}} & n = 1 \\ \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{solving of subproblems}} + \underbrace{\Theta(n)}_{\text{combination of solutions}} & n > 1 \end{cases}$$

Divide and Conquer

Maximum Subtotal - Number of steps $T(n)$

Recursion equation:

$$T(n) = \begin{cases} \underbrace{\Theta(1)}_{\text{trivial case}} & n = 1 \\ \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{solving of subproblems}} + \underbrace{\Theta(n)}_{\text{combination of solutions}} & n > 1 \end{cases}$$

- There exist two constants a and b with:

$$T(n) \leq \begin{cases} a & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n & n > 1 \end{cases}$$

Divide and Conquer

Maximum Subtotal - Number of steps $T(n)$

Recursion equation:

$$T(n) = \begin{cases} \underbrace{\Theta(1)}_{\text{trivial case}} & n = 1 \\ \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{solving of subproblems}} + \underbrace{\Theta(n)}_{\text{combination of solutions}} & n > 1 \end{cases}$$

- There exist two constants a and b with:

$$T(n) \leq \begin{cases} a & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n & n > 1 \end{cases}$$

- We define $c := \max(a, b)$:

$$T(n) \leq \begin{cases} c & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n & n > 1 \end{cases}$$

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

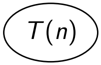
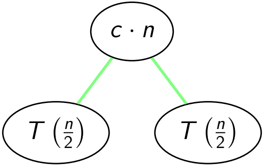


Figure: Illustration of the runtime

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

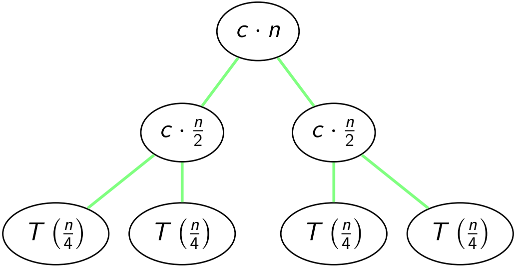


$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

Figure: Illustration of the runtime

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$



$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}$$

Figure: Illustration of the runtime

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

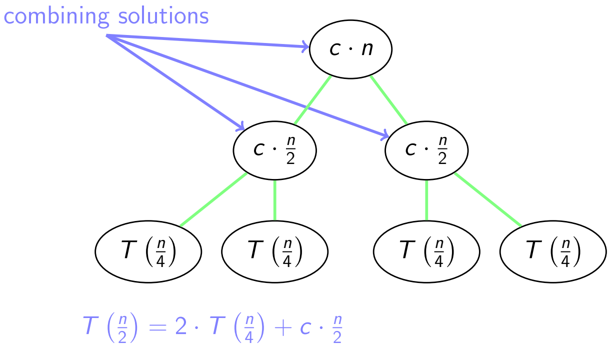


Figure: Illustration of the runtime

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

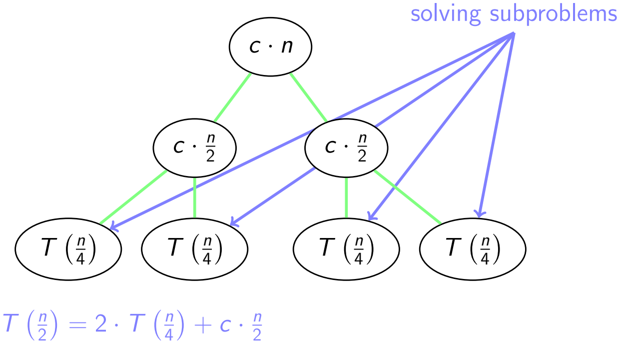
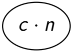


Figure: Illustration of the runtime

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

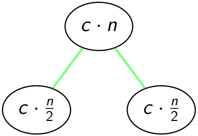


1 node processing n elements
$\Rightarrow c \cdot n$

Figure: Recursion tree method

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$



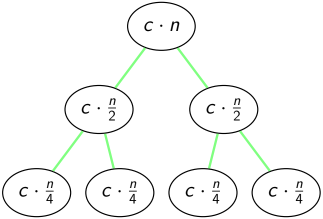
1 node processing n elements
$\Rightarrow c \cdot n$

2 nodes processing $\frac{n}{2}$ elements
$\Rightarrow 2\,c \cdot \frac{n}{2} = c \cdot n$

Figure: Recursion tree method

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$



1 node processing n elements
$\Rightarrow c \cdot n$

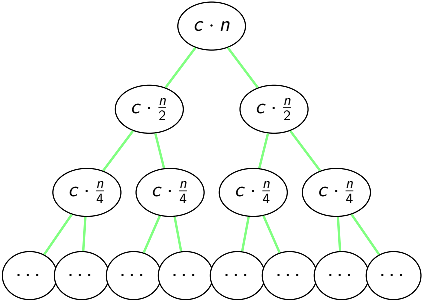
2 nodes processing $\frac{n}{2}$ elements
$\Rightarrow 2\,c \cdot \frac{n}{2} = c \cdot n$

4 nodes processing $\frac{n}{4}$ elements
$\Rightarrow 4\,c \cdot \frac{n}{4} = c \cdot n$

Figure: Recursion tree method

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$



1 node processing n elements
$\Rightarrow c \cdot n$

2 nodes processing $\frac{n}{2}$ elements
$\Rightarrow 2\,c \cdot \frac{n}{2} = c \cdot n$

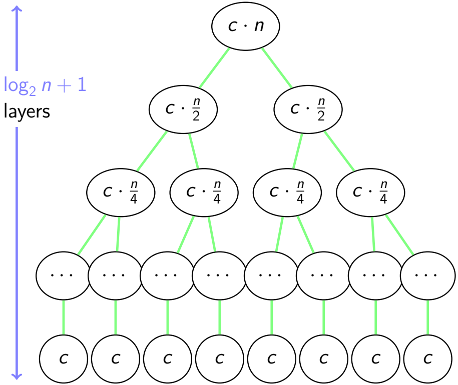
4 nodes processing $\frac{n}{4}$ elements
$\Rightarrow 4\,c \cdot \frac{n}{4} = c \cdot n$

2^i nodes processing $\frac{n}{2^i}$ elements
$\Rightarrow 2^i\,c \cdot \frac{n}{2^i} = c \cdot n$

Figure: Recursion tree method

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$



1 node processing n elements
$\Rightarrow c \cdot n$

2 nodes processing $\frac{n}{2}$ elements
$\Rightarrow 2\,c \cdot \frac{n}{2} = c \cdot n$

4 nodes processing $\frac{n}{4}$ elements
$\Rightarrow 4\,c \cdot \frac{n}{4} = c \cdot n$

2^i nodes processing $\frac{n}{2^i}$ elements
$\Rightarrow 2^i\,c \cdot \frac{n}{2^i} = c \cdot n$

n nodes processing 1 element
$\Rightarrow c \cdot n$

Figure: Recursion tree method

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

Depth:

Depth:

- Top level with depth $i = 0$

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

Depth:

- Top level with depth $i = 0$
- Lowest level with $2^i = n$ elements

$$\Rightarrow i = \log_2 n$$

Divide and Conquer

Maximum Subtotal - Illustration of $T(n)$

Depth:

- Top level with depth $i = 0$
- Lowest level with $2^i = n$ elements

$$\Rightarrow i = \log_2 n$$

Runtime:

Depth:

- Top level with depth $i = 0$
- Lowest level with $2^i = n$ elements

$$\Rightarrow i = \log_2 n$$

Runtime:

- A total of $\log_2 n + 1$ levels with each cost of $c \cdot n$
  The costs of merging the solutions and solving of the trivial problems are the same here

Depth:

- Top level with depth $i = 0$
- Lowest level with $2^i = n$ elements

$$\Rightarrow i = \log_2 n$$

Runtime:

- A total of $\log_2 n + 1$ levels with each cost of $c \cdot n$
  The costs of merging the solutions and solving of the trivial problems are the same here

$$T(n) = c \cdot n \log_2 n + c \cdot n \in \Theta(n \log n)$$

Summary:

Summary:

- Direct solution is slow with $\mathcal{O}(n^3)$

Divide and Conquer

Maximum Subtotal - Summary

Summary:

- Direct solution is slow with $\mathcal{O}(n^3)$
- Better solution with incremental update of sum was $\mathcal{O}(n^2)$

Divide and Conquer

Maximum Subtotal - Summary

Summary:

- Direct solution is slow with $\mathcal{O}(n^3)$
- Better solution with incremental update of sum was $\mathcal{O}(n^2)$
- Divide and conquer approach results in $\mathcal{O}(n \log n)$

Summary:

- Direct solution is slow with $\mathcal{O}(n^3)$
- Better solution with incremental update of sum was $\mathcal{O}(n^2)$
- Divide and conquer approach results in $\mathcal{O}(n \log n)$
- There is an approach running in $\mathcal{O}(n)$ if you assume that all subtotals are positive

Divide and Conquer

Maximum Subtotal

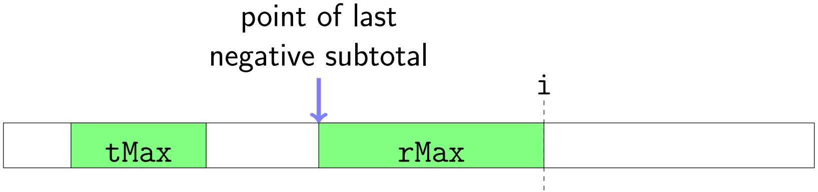


Figure: Scanning the array in linear time

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation - linear runtime
def maxSubArray(X):
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation - linear runtime
def maxSubArray(X):
    # sum, start index
    rMax, irMax = 0, 0 # current maximum
    tMax, itMax = 0, 0 # total maximum
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation - linear runtime
def maxSubArray(X):
    # sum, start index
    rMax, irMax = 0, 0 # current maximum
    tMax, itMax = 0, 0 # total maximum

    for i in range(len(X)):
        if rMax == 0:
            irMax = i
        rMax = max(0, rMax + X[i])
```

Divide and Conquer

Maximum Subtotal - Python

```
#Implementation - linear runtime
def maxSubArray(X):
    # sum, start index
    rMax, irMax = 0, 0 # current maximum
    tMax, itMax = 0, 0 # total maximum

    for i in range(len(X)):
        if rMax == 0:
            irMax = i
        rMax = max(0, rMax + X[i])

        if rMax > tMax:
            tMax, itMax = rMax, irMax

    return (tMax, itMax)
```

Structure

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} \overbrace{f_0(n)}^{\text{trivial case for } n_0} & n = n_0 \\ \underbrace{a \cdot T\left(\frac{n}{b}\right)}_{\substack{\text{solving of } a \\ \text{subproblems} \\ \text{with reduced} \\ \text{input size } \frac{n}{b}}} + \underbrace{f(n)}_{\substack{\text{slicing and} \\ \text{splicing of} \\ \text{subsolutions}}} & n > n_0 \end{cases}$$

Recursion equation:

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

- n_0 is normally small, $f_0(n_0) \in \Theta(1)$

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

- n_0 is normally small, $f_0(n_0) \in \Theta(1)$
- Normally $a > 1$ and $b > 1$

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

- n_0 is normally small, $f_0(n_0) \in \Theta(1)$
- Normally $a > 1$ and $b > 1$
- Dependent on the strategy of solving $T(n)$ f_0 is ignored

Recursion equation:

- Describes the runtime for recursive functions:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

- n_0 is normally small, $f_0(n_0) \in \Theta(1)$
- Normally $a > 1$ and $b > 1$
- Dependent on the strategy of solving $T(n)$ f_0 is ignored
- $T(n)$ is only defined for integers of $\frac{n}{b}$ which is often ignored in benefit of a simpler solution

Structure

Substitution Method:

Substitution Method:

- Guess the solution and prove it with induction

Substitution Method:

- Guess the solution and prove it with induction
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Substitution Method:

- Guess the solution and prove it with induction
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

- Assumption: $T(n) = n + n \cdot \log_2 n$

Induction:

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

Recursion Equations
Substitution Method

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$
$$\stackrel{IA}{=} 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot \log_2 \frac{n}{2}\right) + n$$

Recursion Equations

Substitution Method

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
&\overset{IA}{=} 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot \log_2 \frac{n}{2}\right) + n \\
&= 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot (\log_2 n - 1)\right) + n
\end{aligned}
$$

Recursion Equations

Substitution Method

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
&\overset{IA}{=} 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot \log_2 \frac{n}{2}\right) + n \\
&= 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot (\log_2 n - 1)\right) + n \\
&= n + n \log_2 n - n + n
\end{aligned}
$$

Recursion Equations

Substitution Method

Induction:

- Induction basis (for $n = 1$): $T(1) = 1 + 1 \cdot \log_2 1 = 1$
- Induction step (from $\frac{n}{2}$ to n):

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
&\overset{IA}{=} 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot \log_2 \frac{n}{2}\right) + n \\
&= 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \cdot (\log_2 n - 1)\right) + n \\
&= n + n \log_2 n - n + n \\
&= n + n \log_2 n
\end{aligned}
$$

Substitution Method:

Substitution Method:

- Alternative assumption

Substitution Method:

- Alternative assumption
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 0 \end{cases}$$

Substitution Method:

- Alternative assumption
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 0 \end{cases}$$

- Assumption: $T(n) \in O(n \log n)$

Substitution Method:

- Alternative assumption
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 0 \end{cases}$$

- Assumption: $T(n) \in O(n \log n)$
- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$

Induction:

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$

Recursion Equations

Substitution Method

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\ &\leq 2 \cdot \left(c \cdot \frac{n}{2} \log_2 \frac{n}{2}\right) + n \end{aligned}$$

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\ &\leq 2 \cdot \left(c \cdot \frac{n}{2} \log_2 \frac{n}{2}\right) + n \\ &= c \cdot n \log_2 n - c \cdot n \log_2 2 + n \end{aligned}$$

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\ &\leq 2 \cdot \left(c \cdot \frac{n}{2} \log_2 \frac{n}{2}\right) + n \\ &= c \cdot n \log_2 n - c \cdot n \log_2 2 + n \\ &= c \cdot n \log_2 n - c \cdot n + n \end{aligned}$$

Induction:

- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$
- Induction step (from $\frac{n}{2}$ to n):

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
&\leq 2 \cdot \left(c \cdot \frac{n}{2} \log_2 \frac{n}{2}\right) + n \\
&= c \cdot n \log_2 n - c \cdot n \log_2 2 + n \\
&= c \cdot n \log_2 n - c \cdot n + n \\
&\leq c \cdot n \log_2 n, \quad c \geq 1
\end{aligned}
$$

Structure

Recursion tree method:

Recursion tree method:

- Can be used to make assumptions about the runtime

Recursion tree method:

- Can be used to make assumptions about the runtime
- Example:

$$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + \Theta(n^2) \leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$$

Recursion Equations

Recursion Tree Method

$T\left(n\right) = 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$

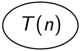
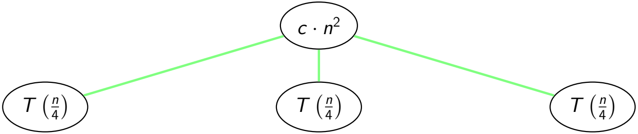


Figure: Recursion tree of example

Recursion Equations

Recursion Tree Method

$T\left(n\right) = 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$



$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$

Figure: Recursion tree of example

Recursion Equations

Recursion Tree Method

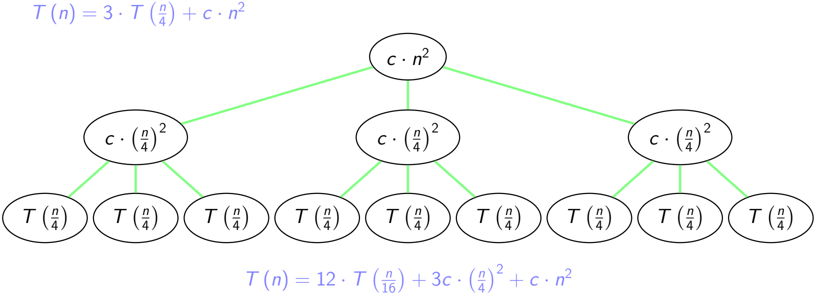


Figure: Recursion tree of example

Recursion Equations

Recursion Tree Method

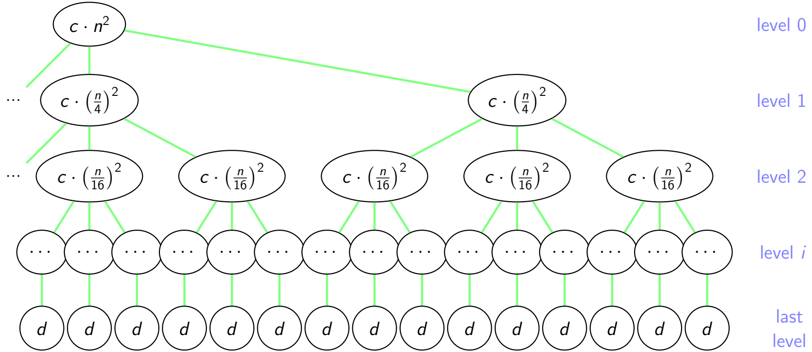


Figure: Levels of the recursion tree

Recursion Equations

Recursion Tree Method Costs

Costs of connecting the partial solutions:

(excludes the last layer)

Recursion Equations

Recursion Tree Method Costs

Costs of connecting the partial solutions:
(excludes the last layer)

- Size of partial problems on level i: $\mathrm{s}_i(n) = \left(\frac{1}{4}\right)^i \cdot n$

Costs of connecting the partial solutions:
(excludes the last layer)

- Size of partial problems on level i: $\mathrm{s}_i(n) = \left(\frac{1}{4}\right)^i \cdot n$
- Costs of partial problem on level i:

$$T_{i_p}(n) = c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2$$

Costs of connecting the partial solutions:
(excludes the last layer)

- Size of partial problems on level i: $\mathrm{s}_i(n) = \left(\frac{1}{4}\right)^i \cdot n$
- Costs of partial problem on level i:

$$T_{i_p}(n) = c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2$$

- Number of partial problems on level i: $n_i = 3^i$

Costs of connecting the partial solutions:

(excludes the last layer)

- Size of partial problems on level i: $\mathrm{s}_i(n) = \left(\frac{1}{4}\right)^i \cdot n$
- Costs of partial problem on level i:

$$T_{i_p}(n) = c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2$$

- Number of partial problems on level i: $n_i = 3^i$
- Costs on level i:

$$T_i(n) = 3^i \cdot c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2 = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$$

Costs of solving partial solutions: (only the last layer)

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$

Recursion Equations

Recursion Tree Method Costs

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$
- Costs of partial problem on the last level: $T_{i+1_p}(n) = d$

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$
- Costs of partial problem on the last level: $T_{i+1_p}(n) = d$
- With this the depth of the tree is:

$$\left(\tfrac{1}{4}\right)^i \cdot n = 1 \qquad \Rightarrow n = 4^i \qquad \Rightarrow i = \log_4 n$$

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$
- Costs of partial problem on the last level: $T_{i+1_p}(n) = d$
- With this the depth of the tree is:

$$\left(\frac{1}{4}\right)^i \cdot n = 1 \qquad \Rightarrow n = 4^i \qquad \Rightarrow i = \log_4 n$$

- Number of partial problems on the last level:

$$n_{i+1} = 3^{\log_4 n}$$

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$
- Costs of partial problem on the last level: $T_{i+1_p}(n) = d$
- With this the depth of the tree is:

$$\left(\tfrac{1}{4}\right)^i \cdot n = 1 \qquad \Rightarrow n = 4^i \qquad \Rightarrow i = \log_4 n$$

- Number of partial problems on the last level:

$$n_{i+1} = 3^{\log_4 n} \quad = n^{\log_4 3} \quad \leftarrow \text{next slide}$$

Costs of solving partial solutions: (only the last layer)

- Size of partial problems on the last level: $\mathrm{s}_{i+1}(n) = 1$
- Costs of partial problem on the last level: $T_{{i+1}_p}(n) = d$
- With this the depth of the tree is:

$$\left(\frac{1}{4}\right)^i \cdot n = 1 \qquad \Rightarrow n = 4^i \qquad \Rightarrow i = \log_4 n$$

- Number of partial problems on the last level:

$$n_{i+1} = 3^{\log_4 n} \quad = n^{\log_4 3} \quad \leftarrow \text{next slide}$$

- Costs on the last level: $T_{i+1}(n) = d \cdot n^{\log_4 3}$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\log_4 n = \log_4 \left(3^{\log_3 n}\right) \qquad \text{uses } n = 3^{\log_3 n}$$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$
\begin{aligned}
\log_4 n &= \log_4\left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\
&= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a
\end{aligned}
$$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\begin{aligned} \log_4 n &= \log_4 \left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\ &= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a \end{aligned}$$

- This proves the general log rule $\log_b c = \log_a c \cdot \log_b a$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\begin{aligned} \log_4 n &= \log_4 \left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\ &= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a \end{aligned}$$

- This proves the general log rule $\log_b c = \log_a c \cdot \log_b a$
- Now the whole expression:

$$3^{\log_4 n} = 3^{\log_3 n \cdot \log_4 3} \qquad \text{uses reformulation above}$$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\begin{aligned} \log_4 n &= \log_4 \left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\ &= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a \end{aligned}$$

- This proves the general log rule $\log_b c = \log_a c \cdot \log_b a$
- Now the whole expression:

$$\begin{aligned} 3^{\log_4 n} &= 3^{\log_3 n \cdot \log_4 3} && \text{uses reformulation above} \\ &= \left(3^{\log_3 n}\right)^{\log_4 3} && \text{uses } x^{a \cdot b} = (x^a)^b \end{aligned}$$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\begin{aligned} \log_4 n &= \log_4 \left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\ &= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a \end{aligned}$$

- This proves the general log rule $\log_b c = \log_a c \cdot \log_b a$
- Now the whole expression:

$$\begin{aligned} 3^{\log_4 n} &= 3^{\log_3 n \cdot \log_4 3} && \text{uses reformulation above} \\ &= \left(3^{\log_3 n}\right)^{\log_4 3} && \text{uses } x^{a \cdot b} = (x^a)^b \\ &= n^{\log_4 3} \end{aligned}$$

Fun with logarithm

Logarithm

- Transforming $3^{\log_4 n}$ uses general log rules

$$\begin{aligned} \log_4 n &= \log_4 \left(3^{\log_3 n}\right) && \text{uses } n = 3^{\log_3 n} \\ &= \log_3 n \cdot \log_4 3 && \text{uses } \log a^b = b \cdot \log a \end{aligned}$$

- This proves the general log rule $\log_b c = \log_a c \cdot \log_b a$
- Now the whole expression:

$$\begin{aligned} 3^{\log_4 n} &= 3^{\log_3 n \cdot \log_4 3} && \text{uses reformulation above} \\ &= \left(3^{\log_3 n}\right)^{\log_4 3} && \text{uses } x^{a \cdot b} = (x^a)^b \\ &= n^{\log_4 3} \end{aligned}$$

- This term will recur in the master theorem

Total costs:

Recursion Equations

Total costs

Total costs:

- Costs of level i: $T_i(n) = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$

Recursion Equations

Total costs

Total costs:

- Costs of level i: $T_i(n) = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$
- Costs of last level: $T_{i+1}(n) = d \cdot n^{\log_4 3}$

Total costs:

- Costs of level i: $T_i(n) = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$
- Costs of last level: $T_{i+1}(n) = d \cdot n^{\log_4 3}$

$$T(n) = \underbrace{\sum_{i=0}^{(\log_4 n)-1} \left(\tfrac{3}{16}\right)^i \cdot c \cdot n^2}_{\substack{\text{geometric series,}\\ \text{constant}\\ \left(\text{even with infinite elements}\right)}} + \underbrace{d \cdot n^{\log_4 3}}_{\substack{\log_4 3 < 1,\\ \text{grows a lot}\\ \text{slower than } n^2}} \in \mathcal{O}(n^2)$$

Recursion Equations

Total costs

Total costs:

- Costs of level i: $T_i(n) = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$
- Costs of last level: $T_{i+1}(n) = d \cdot n^{\log_4 3}$

$$T(n) = \underbrace{\sum_{i=0}^{(\log_4 n)-1} \left(\tfrac{3}{16}\right)^i \cdot c \cdot n^2}_{\substack{\text{geometric series,}\\ \text{constant}\\ \left(\text{even with infinite elements}\right)}} + \underbrace{d \cdot n^{\log_4 3}}_{\substack{\log_4 3 < 1,\\ \text{grows a lot}\\ \text{slower than } n^2}} \in \mathcal{O}(n^2)$$

- Here: The costs of connecting the partial problems dominate

Recursion Equations

Geometric Series

- **Geometric progression:**
  Quotient of two neighboring sequence parts is constant

$$2^0, 2^1, 2^2, \ldots, 2^k$$

- **Geometric progression:**
  Quotient of two neighboring sequence parts is constant

$$2^0, 2^1, 2^2, \ldots, 2^k$$

- **Geometric series:**
  The series (cumulative sum) of a geometric sequence

Recursion Equations

Geometric Series

- **Geometric progression:**
  Quotient of two neighboring sequence parts is constant

$$2^0, 2^1, 2^2, \ldots, 2^k$$

- **Geometric series:**
  The series (cumulative sum) of a geometric sequence
- For $\mid q \mid < 1$:

$$\sum_{k=0}^{\infty} a_0 \cdot q^k = \frac{a_0}{1-q} \qquad \Rightarrow \text{constant}$$

Proof of $\mathcal{O}(n^2)$:

Proof of $\mathcal{O}(n^2)$:

- We know:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{4}\right) + \Theta(n^2) \\ &\leq 3T\left(\frac{n}{4}\right) + c \cdot n^2 \end{aligned}$$

Proof of $\mathcal{O}(n^2)$:

- We know:

$$T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$$
$$\leq 3T\left(\frac{n}{4}\right) + c \cdot n^2$$

- Assumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) \leq k \cdot n^2$$

Proof of $\mathcal{O}(n^2)$:

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

- Substitution method:

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

- Substitution method:

$$T(n) \leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$$

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

- Substitution method:

$$\begin{aligned} T(n) &\leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2 \\ &\leq 3\,k \cdot \left(\frac{n}{4}\right)^2 + c \cdot n^2 \end{aligned}$$

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

- Substitution method:

$$\begin{aligned} T(n) &\leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2 \\ &\leq 3\,k \cdot \left(\frac{n}{4}\right)^2 + c \cdot n^2 \\ &= \frac{3}{16}\,k \cdot n^2 + c \cdot n^2 \end{aligned}$$

Proof of $\mathcal{O}(n^2)$:

- Presumption: $T(n) \in \mathcal{O}(n^2)$, so there exists a $k > 0$ with

$$T(n) < k \cdot n^2$$

- Substitution method:

$$\begin{aligned} T(n) &\leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2 \\ &\leq 3\,k \cdot \left(\frac{n}{4}\right)^2 + c \cdot n^2 \\ &= \frac{3}{16}\,k \cdot n^2 + c \cdot n^2 \\ &\leq k \cdot n^2 && \text{for } k \geq \frac{16}{13}\,c \end{aligned}$$

Structure

Master theorem:

Master theorem:

- Approach to solve for a recursion equation of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

Recursion Equations

Master theorem

Master theorem:

- Approach to solve for a recursion equation of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- $T(n)$ is the runtime of an algorithm ...

Master theorem:

- Approach to solve for a recursion equation of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- $T(n)$ is the runtime of an algorithm ...
  - ... which divides a problem of size n in a partial problems

Master theorem:

- Approach to solve for a recursion equation of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- $T(n)$ is the runtime of an algorithm ...
  - ... which divides a problem of size n in a partial problems
  - ... which solves each partial problem recursively with a runtime of $T\left(\frac{n}{b}\right)$

Master theorem:

- Approach to solve for a recursion equation of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- $T(n)$ is the runtime of an algorithm ...
  - ... which divides a problem of size n in a partial problems
  - ... which solves each partial problem recursively with a runtime of $T\left(\frac{n}{b}\right)$
  - ... which takes $f(n)$ steps to merge all partial solutions

Master theorem:

Master theorem:

- In the examples we have seen that . . .

Master theorem:

- In the examples we have seen that ...
  - Either the runtime of connecting the solutions dominates

Master theorem:

- In the examples we have seen that ...
  - Either the runtime of connecting the solutions dominates
  - Or the runtime of solving the problems dominates

Master theorem:

- In the examples we have seen that ...
  - Either the runtime of connecting the solutions dominates
  - Or the runtime of solving the problems dominates
  - Or both have equal influence on runtime

Master theorem:

- In the examples we have seen that ...
  - Either the runtime of connecting the solutions dominates
  - Or the runtime of solving the problems dominates
  - Or both have equal influence on runtime
- **Simple form:** Special case with runtime of connecting the solutions $f(n) \in O(n)$

Simple form:

Recursion Equations

Master theorem (Simple Form)

Simple form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \underbrace{c \cdot n}_{\substack{\text{Is any } f(n) \\ \text{in general form}}}, \qquad a \geq 1, b > 1, c > 0$$

Simple form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \underbrace{c \cdot n}_{\substack{\text{Is any } f(n) \\ \text{in general form}}}, \qquad a \geq 1, b > 1, c > 0$$

- This yields a runtime of:

Recursion Equations

Master theorem (Simple Form)

Simple form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \underbrace{c \cdot n}_{\substack{\text{Is any } f(n) \\ \text{in general form}}}, \qquad a \geq 1, b > 1, c > 0$$

- This yields a runtime of:

$$T(n) = \begin{cases} \Theta(\overbrace{n^{\log_b a}}^{\text{Number of leaves}}) & \text{if } a > b \\ \Theta(n \log n) & \text{if } a = b \\ \Theta(n) & \text{if } a < b \end{cases}$$

Recursion Equations

Master theorem (Simple Form)

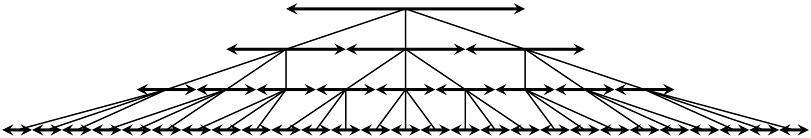


Figure: Simple recursion equation with $a = 3, b = 2$

Recursion Equations

Master theorem (Simple Form)

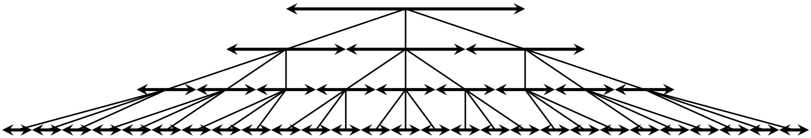


Figure: Simple recursion equation with $a = 3, b = 2$

Case 1: $a > b$

Recursion Equations

Master theorem (Simple Form)

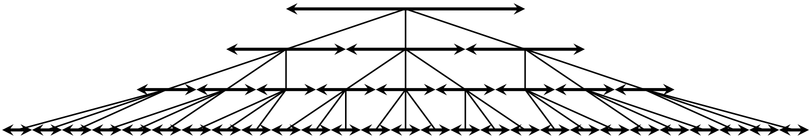


Figure: Simple recursion equation with $a = 3, b = 2$

Case 1: $a > b$

- Three partial problems with $\frac{1}{2}$ the size

Recursion Equations

Master theorem (Simple Form)

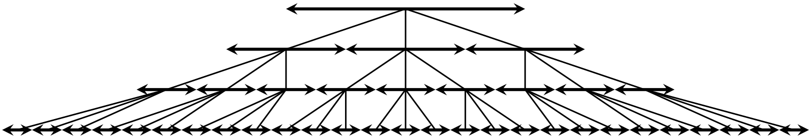


Figure: Simple recursion equation with $a = 3, b = 2$

Case 1: $a > b$

- Three partial problems with $\frac{1}{2}$ the size
- Solving the partial problems dominates (last layer, leaves)

Recursion Equations

Master theorem (Simple Form)

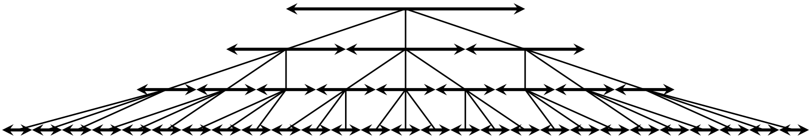


Figure: Simple recursion equation with $a = 3, b = 2$

Case 1: $a > b$

- Three partial problems with $\frac{1}{2}$ the size
- Solving the partial problems dominates (last layer, leaves)
- Runtime of $\Theta(n^{\log_b a})$

Recursion Equations

Master theorem (Simple Form)

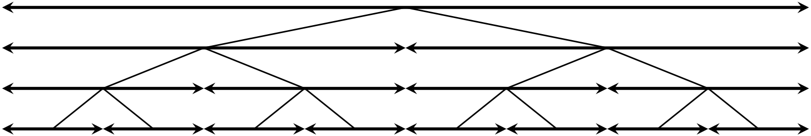


Figure: Simple recursion equation with $a = 2, b = 2$

Recursion Equations

Master theorem (Simple Form)

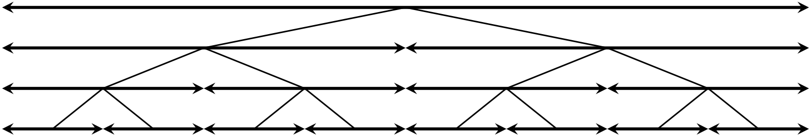


Figure: Simple recursion equation with $a = 2, b = 2$

Case 2: $a = b$

Recursion Equations

Master theorem (Simple Form)

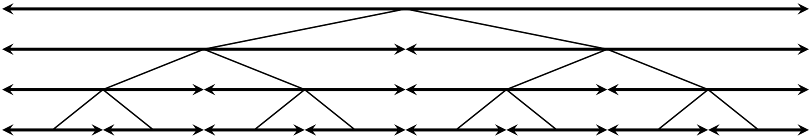


Figure: Simple recursion equation with $a = 2, b = 2$

Case 2: $a = b$

- Two partial problems with $\frac{1}{2}$ the size

Recursion Equations

Master theorem (Simple Form)

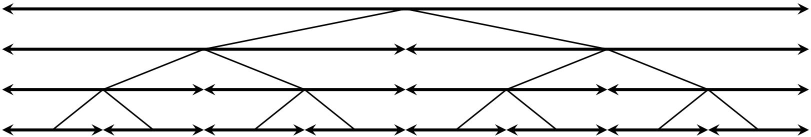


Figure: Simple recursion equation with $a = 2, b = 2$

Case 2: $a = b$

- Two partial problems with $\frac{1}{2}$ the size
- Each layer has equal costs, $\log n$ layers

Recursion Equations

Master theorem (Simple Form)

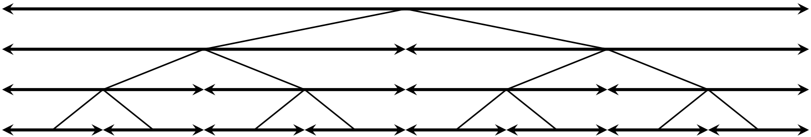


Figure: Simple recursion equation with $a = 2, b = 2$

Case 2: $a = b$

- Two partial problems with $\frac{1}{2}$ the size
- Each layer has equal costs, $\log n$ layers
- Runtime of $\Theta(n \log n)$

Recursion Equations

Master theorem (Simple Form)

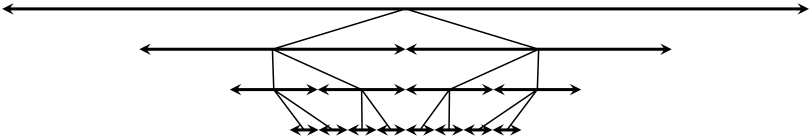


Figure: Simple recursion equation with $a = 2, b = 3$

Recursion Equations

Master theorem (Simple Form)

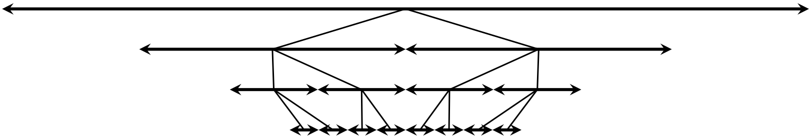


Figure: Simple recursion equation with $a = 2, b = 3$

Case 3: $a < b$

Recursion Equations

Master theorem (Simple Form)

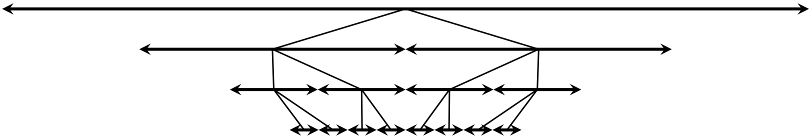


Figure: Simple recursion equation with $a = 2, b = 3$

Case 3: $a < b$

- Two partial problems with $\frac{1}{3}$ the size

Recursion Equations

Master theorem (Simple Form)

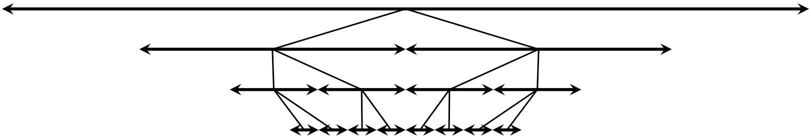


Figure: Simple recursion equation with $a = 2, b = 3$

Case 3: $a < b$

- Two partial problems with $\frac{1}{3}$ the size
- Connecting all partial solutions dominates (first layer, root)

Recursion Equations

Master theorem (Simple Form)

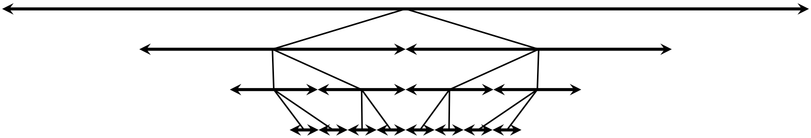


Figure: Simple recursion equation with $a = 2, b = 3$

Case 3: $a < b$

- Two partial problems with $\frac{1}{3}$ the size
- Connecting all partial solutions dominates (first layer, root)
- Runtime of $\Theta(n)$

Recursion Equations

Master theorem (Simple Form)

For a recursion equation like

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n, \qquad a \geq 1, b > 1, c > 0$$

Recursion Equations

Master theorem (Simple Form)

For a recursion equation like

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n, \qquad a \geq 1, b > 1, c > 0$$

- ... yields to a runtime of:

Recursion Equations

Master theorem (Simple Form)

For a recursion equation like

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n, \qquad a \geq 1, b > 1, c > 0$$

- ... yields to a runtime of:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b \\ \Theta(n \log_b n) & \text{if } a = b \\ \Theta(n) & \text{if } a < b \end{cases}$$

Recursion Equations

Master theorem (Simple Form)

For a recursion equation like

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n, \qquad a \geq 1, b > 1, c > 0$$

- ... yields to a runtime of:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b \\ \Theta(n \log_b n) & \text{if } a = b \\ \Theta(n) & \text{if } a < b \end{cases}$$

- Proof with *geometric series*: Number of operations per layer grows / shrinks by constant factor $\frac{a}{b}$

Structure

Master theorem (general form):

Master theorem (general form):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

Master theorem (general form):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- **Case 1:** $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$
  Solving the partial problems dominates
  (last layer, leaves)

Master theorem (general form):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad a \geq 1, b > 1$$

- **Case 1:** $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$
  Solving the partial problems dominates
  (last layer, leaves)

- **Case 2:** $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$
  Each layer has equal costs, $\log_b n$ layers

Master theorem (general form):

- **Case 3:** $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$
  Connecting all partial solutions in first layer (root) dominates

  Regularity condition:

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), \qquad 0 \leq c \leq 1,$$
$$n > n_0$$

Recursion Equations

Master theorem (General Form) - Case 1

Case 1 - Example: if $f(n) \in O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$

Solving the partial problems dominates (last layer, leaves)

Recursion Equations

Master theorem (General Form) - Case 1

Case 1 - Example: $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$

Solving the partial problems dominates (last layer, leaves)

Recursion Equations

Master theorem (General Form) - Case 1

Case 1 - Example: $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$

Solving the partial problems dominates (last layer, leaves)

- $T(n) = 8 \cdot T(\frac{n}{2}) + 1000 \cdot n^2$

$$a = 8,\ b = 2,\ f(n) = 1000 \cdot n^2,\ \underbrace{\log_b a = \log_2 8 = 3}_{n^3 \text{ leaves}}$$

$$f(n) \in \mathcal{O}(n^{3-\varepsilon}) \Rightarrow T(n) \in \Theta(n^3)$$

Recursion Equations

Master theorem (General Form) - Case 1

Case 1 - Example: $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$

Solving the partial problems dominates (last layer, leaves)

- $T(n) = 8 \cdot T(\frac{n}{2}) + 1000 \cdot n^2$

  $$a = 8,\ b = 2,\ f(n) = 1000 \cdot n^2,\ \underbrace{\log_b a = \log_2 8 = 3}_{n^3 \text{ leaves}}$$

  $$f(n) \in \mathcal{O}(n^{3-\varepsilon}) \Rightarrow T(n) \in \Theta(n^3)$$

- $T(n) = 9 \cdot T(\frac{n}{3}) + 17 \cdot n$

  $$a = 9,\ b = 3,\ f(n) = 17 \cdot n,\ \underbrace{\log_b a = \log_3 9 = 2}_{n^2 \text{ leaves}}$$

  $$f(n) \in \mathcal{O}(n^{2-\varepsilon}) \Rightarrow T(n) \in \Theta(n^2)$$

Recursion Equations

Master theorem (General Form) - Case 2

Case 2: if $f(n) \in \Theta(n^{\log_b a})$

Each layer has equal costs, $\log n$ layers

Recursion Equations

Master theorem (General Form) - Case 2

Case 2: $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$

Each layer has equal costs, $\log n$ layers

Recursion Equations

Master theorem (General Form) - Case 2

Case 2: $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$

Each layer has equal costs, $\log n$ layers

- $T(n) = 2 \cdot T(\frac{n}{2}) + 10 \cdot n$

$$a = 2,\ b = 2,\ f(n) = 10 \cdot n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

$f(n) \in \Theta(n^{\log_2 2}) \Rightarrow T(n) \in \Theta(n \log n)$

Recursion Equations

Master theorem (General Form) - Case 2

Case 2: $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$

Each layer has equal costs, $\log n$ layers

- $T(n) = 2 \cdot T(\frac{n}{2}) + 10 \cdot n$

$$a = 2,\ b = 2,\ f(n) = 10 \cdot n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

$$f(n) \in \Theta(n^{\log_2 2}) \Rightarrow T(n) \in \Theta(n \log n)$$

- $T(n) = T(\frac{2n}{3}) + 1$

$$a = 1,\ b = \frac{2}{3},\ f(n) = 1,\ \underbrace{\log_b a = \log_{3/2} 1 = 0}_{n^0 \text{ leaves} = 1 \text{ leaf}}$$

$$f(n) \in \Theta(n^{\log_{3/2} 1}) \Rightarrow T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$$

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

- $T(n) = 2 \cdot T(\frac{n}{2}) + n^2$

$$a = 2,\ b = 2,\ f(n) = n^2,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

$f(n) \in \Omega(n^{1+\varepsilon})$

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

- $T(n) = 2 \cdot T(\frac{n}{2}) + n^2$

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

- $T(n) = 2 \cdot T(\frac{n}{2}) + n^2$

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

- $T(n) = 2 \cdot T(\frac{n}{2}) + n^2$
- $f(n) \in \Omega(n^{1+\varepsilon})$

Recursion Equations

Master theorem (General Form) - Case 3

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions in first layer (root) dominates

- $T(n) = 2 \cdot T(\frac{n}{2}) + n^2$
- $f(n) \in \Omega(n^{1+\varepsilon})$
- Check if regularity condition also holds:

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

$$2 \cdot \left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \qquad \Rightarrow \frac{1}{2} \cdot n^2 \leq c \cdot n^2 \qquad \Rightarrow c \geq \frac{1}{2}$$

$\Rightarrow T(n) \in \Theta(n^2)$

Master theorem:

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

$$a = 2,\ b = 2,\ f(n) = n \log n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

Recursion Equations

Master theorem (General Form)

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

$$a = 2,\ b = 2,\ f(n) = n \log n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

- **Case 1:** $f(n) \notin O(n^{1-\varepsilon})$

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

$$a = 2,\ b = 2,\ f(n) = n \log n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

- **Case 1:** $f(n) \notin O(n^{1-\varepsilon})$
- **Case 2:** $f(n) \notin \Theta(n^1)$

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

$$a = 2,\ b = 2,\ f(n) = n \log n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

- **Case 1:** $f(n) \notin O(n^{1-\varepsilon})$
- **Case 2:** $f(n) \notin \Theta(n^1)$
- **Case 3:** $f(n) \notin \Omega(n^{1+\varepsilon})$

Master theorem:

- Not always applicable: $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

$$a = 2,\ b = 2,\ f(n) = n \log n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

- **Case 1:** $f(n) \notin O(n^{1-\varepsilon})$
- **Case 2:** $f(n) \notin \Theta(n^1)$
- **Case 3:** $f(n) \notin \Omega(n^{1+\varepsilon})$

$n \log n$ is *asymptotically* larger than n,
but not *polynominal* larger

Master theorem:

Master theorem:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

Recursion Equations

Master theorem - Summary

Master theorem:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- Three cases depending on the dominance of the terms

Master theorem:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- Three cases depending on the dominance of the terms
- **Case 1:** Solving the partial problems is *polynominal* bigger than merging all solutions
  $T(n) \in \Theta(n^{\log_b a})$, $T(n) \in \Theta(\text{number of leaves})$

Master theorem:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- Three cases depending on the dominance of the terms
- **Case 1:** Solving the partial problems is *polynominal* bigger than merging all solutions
  $T(n) \in \Theta(n^{\log_b a})$, $T(n) \in \Theta(\text{number of leaves})$
- **Case 2:** Each layer has equal costs
  $T(n) \in \Theta(n^{\log_b a} \log n)$, $\log n$ layers

Master theorem:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- Three cases depending on the dominance of the terms
- **Case 1:** Solving the partial problems is *polynominal* bigger than merging all solutions
  $T(n) \in \Theta(n^{\log_b a})$, $T(n) \in \Theta(\text{number of leaves})$
- **Case 2:** Each layer has equal costs
  $T(n) \in \Theta(n^{\log_b a} \log n)$, $\log n$ layers
- **Case 3:** Connecting all partial solutions is *polynominal* bigger than solving all partial problems
  $T(n) \in \Theta(f(n))$

Further Literature

- **General**

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.
Introduction to Algorithms.
MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.
Algorithms and data structures, 2008.
https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf.

Further Literature

- **Master theorem**

  [Wik] Master theorem
  https://en.wikipedia.org/wiki/Master_theorem