Data Structures and Algorithms (ECEG 4171)

Chapter Two
Recurrences

Introduction

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Recursive algorithms call themselves recursively one or more times to deal with closely related subproblems.
- Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:
  - If the given instance of the problem is small or simple enough, just solve it.
  - Otherwise, reduce the problem to one or more simpler instances of the same problem.

Introduction

- Two well-known algorithms design techniques:
  - Divide-and-conquer algorithms
  - Decrease-and-conquer algorithms
- Divide-and-conquer algorithms are efficient algorithms which follow the following three steps at each level of recursion;
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.

The Factorial Function: Example

- To demonstrate the mechanics of recursion, we begin with computing the value of the factorial function. For any $n \geq 0$:

$$n! = \begin{cases} 1 & , \quad if\ n = 0 \\ n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1 & , \quad if\ n \geq 1 \end{cases}$$

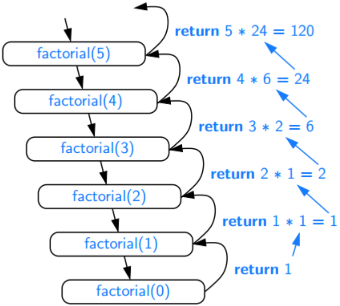
- In recursive definition of a factorial can be given as:

$$n! = \begin{cases} 1 & , \quad if\ n = 0 \\ n \cdot (n-1)! & , \quad if\ n \geq 1 \end{cases}$$

- A recursive definition has two cases:
  1. One or more **base cases**, which refer to fixed values of the function. The above definition has one base case stating that $n! = 1$ for $n = 0$.
  2. One or more **recursive cases**, which define the function in terms of itself. In the above definition, there is one recursive case, which indicates that $n! = n \cdot (n-1)!$ for $n \geq 1$.

Implementation of the Factorial Function

Factorial(5) **trace** :




```
Factorial(n)
  if n==0
    return 1
  else
    return n*Factorial(n-1)
```

Analyzing The Factorial Function:

- The running time $T(n)$ is given as:

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(1) \\
&= T(n-2) + \Theta(1) + \Theta(1) \\
&\vdots \\
&= \Theta(1) + \cdots + \Theta(1) + \Theta(1) = \sum^{n} \Theta(1) = \Theta(\sum^{n} 1) = \Theta(n)
\end{aligned}
$$

Solving Recurrences

- 3 methods:
  - Substitution method
    - ▷ we guess a bound and then use mathematical induction to prove our guess correct.
  - Recursion tree method
    - ▷ converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.
  - Master method
    - ▷ provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a > 1$, $b > 1$, and $f(n)$ is a given function

Review: Proof by Induction

- Basis: show S(0)
- Hypothesis: assume $S(k)$ holds for arbitrary $k \leq n$
- Step: Show $S(n+1)$ follows.

Review: Proof by Induction

- Basis: show S(0)
- Hypothesis: assume $S(k)$ holds for arbitrary $k \leq n$
- Step: Show $S(n+1)$ follows.

Example: Prove $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

Review: Proof by Induction

- Basis: show S(0)
- Hypothesis: assume $S(k)$ holds for arbitrary $k \leq n$
- Step: Show $S(n+1)$ follows.

Example: Prove $1+2+\cdots+n = \frac{n(n+1)}{2}$

- ▷ Basis:
  - If $n = 0$, then, $0 = \frac{0(0+1)}{2}$
- ▷ Inductive hypothesis:
  - Assume $1+2+\cdots+n = \frac{n(n+1)}{2}$
- ▷ Step (show true for $n+1$):

$$\begin{aligned} 1+2+\cdots+n+n+1 &= (1+2+\cdots+n)+(n+1) \\ &= \frac{n(n+1)}{2}+n+1 = \frac{n(n+1)+2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+1+1)}{2} \end{aligned}$$

Substitution Method

- Comprises the following steps:
  1. Guess the form of the solution.
  2. Verify by induction.
  3. Solve for constants.
- Can be used to obtain either upper or lower bound on a recurrence.
- **Example:** $T(n) = 4T(n/2) + n$ (base case: $T(1) = \Theta(1)$).

Substitution Method

- Comprises the following steps:
  1. Guess the form of the solution.
  2. Verify by induction.
  3. Solve for constants.
- Can be used to obtain either upper or lower bound on a recurrence.
- **Example:** $\mathrm{T(n)} = 4\mathrm{T(n/2)} + \mathrm{n}$ (base case: $\mathrm{T}(1) = \Theta(1)$).
  - Guess $\mathrm{T(n)} = \mathrm{O}(n^3)$
  - Assume $\mathrm{T(k)} \leq \mathrm{c}k^3$ for $k < n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4c(n/2)^3 + n \\
&= \frac{1}{2}cn^3 + n \\
&= cn^3 - (\frac{1}{2}cn^3 - n) \\
&\leq cn^3, \Rightarrow \textit{if } \frac{1}{2}cn^3 - n \geq 0. \textit{ True for } c \geq 2,\ n \geq 1.
\end{aligned}
$$

For the base case: $\mathrm{T}(1)=\Theta(1) \leq c(1)^3$, this is true for sufficiently large c.

$T(n) = 4T(n/2) + n$

Try: $T(n) = O(n^2)$

$T(n) = 4T(n/2) + n$

Try: $T(n) = O(n^2)$

– Assume $T(k) \leq ck^2$ for $k < n$.

$$\begin{aligned} T(n) =& 4T(n/2) + n \\ & \leq 4c(n/2)^2 + n \\ & = cn^2 + n \\ & = cn^2 - (-n) \\ & \nleq cn^2 \end{aligned}$$

$T(n) = 4T(n/2) + n$

Try: $T(n) = O(n^2)$

– Assume $T(k) \leq ck^2$ for $k < n$.

$$\begin{aligned} T(n) =& 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &= cn^2 - (-n) \\ &\not\leq cn^2 \end{aligned}$$

– Assume $T(k) \leq c_1 k^2 - c_2 k$ for $k < n$.

$$\begin{aligned} T(n) =& 4T(n/2) + n \\ &\leq 4\left(c_1(n/2)^2 - c_2(n/2)\right) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &= c_1 n^2 - c_2 n - (c_2 - 1)n \\ &\leq c_1 n^2 - c_2 n, \textit{ for } c_2 - 1 \geq 0,\ c_2 \geq 0. \end{aligned}$$

$T(1) \leq c_1 - c_2 \;\Rightarrow\; T(1) = \Theta(1)$

c_1 should be sufficiently large wrt to c_2.

Show that $T(n) = 2T(\lfloor n/2 \rfloor) + n = O(n \lg n)$.

Show that $T(n) = 2T(\lfloor n/2 \rfloor) + n = O(nlgn)$.

Assume $T(k) \leq klgk$ for $k < n$.

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2\frac{cn}{2} lgn/2 + n \\ &= cn(lgn - 1) + n \\ &= cnlgn - n(c - 1) \\ &\leq cnlgn \; for \; c \geq 1 \end{aligned}$$

Base case: $T(1) \leq c1lg1 = 0$, Wrong!
For $n \geq 2$, $T(2)$ can be the base case in the inductive proof letting $n_0 = 2$.

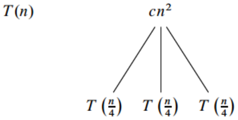
$$T(2) \leq c * 2lg2 = \Theta(1) \; for \; sufficiently \; large \; c.$$

Recrusion-tree Method

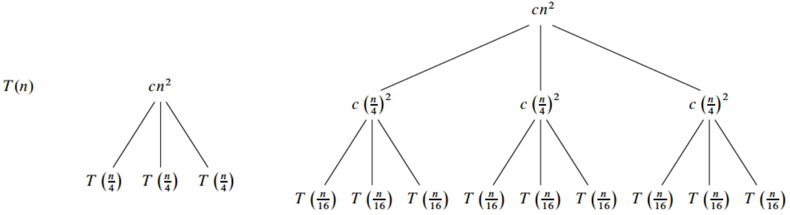
- Convert the recurrence into a tree:
  - ▷ Each node represents the cost of a single subproblem.
  - ▷ Sum the costs within each level.
  - ▷ Then sum all the per-level costs to determine the total cost of all levels of the recursion.
- Best used to generate a good guess for the recurrence which can be verified by substitution method.
- For example, solve $\mathbf{T(n) = 3T(}\lfloor n/4 \rfloor\mathbf{)} + \Theta(n^2)$

$$
\begin{aligned}
T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \\
&= 3T(n/4) + \Theta(n^2)\,(floor\ and\ ceilings\ don't\ matter) \\
&= 3T(n/4) + cn^2,\ c > 0
\end{aligned}
$$

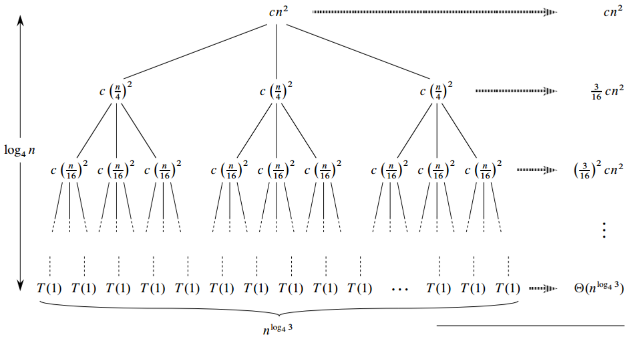
$$
\begin{aligned}
T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \\
&= 3T(n/4) + \Theta(n^2)\,(\textit{floor and ceilings don't matter}) \\
&= 3T(n/4) + cn^2,\ c > 0
\end{aligned}
$$



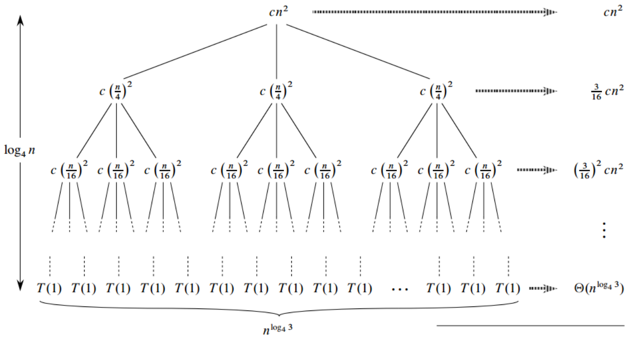
$$
\begin{aligned}
T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \\
&= 3T(n/4) + \Theta(n^2)\,(\textit{floor and ceilings don't matter}) \\
&= 3T(n/4) + cn^2,\; c > 0
\end{aligned}
$$



$T(n) = 3T(n/4) + cn^2$


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\log_4 n$
$T(1)$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

$\mathrm{T(n)} = 3\mathrm{T(n/4)} + \mathrm{c}n^2$



Subproblem size decreases by a factor 4 each time we go down one leve. $\Rightarrow$ at level i, subproblem size is $n/4^{i-1}$. $\Rightarrow$ n = 1, when $n/4^{i-1} = 1 \Rightarrow$ $i = \log_4 n + 1$ levels.

Also, level i has 3^{i-1} nodes $\Rightarrow$ level $\log_4 n + 1$ has $3^{\log_4 n+1-1} = 3^{\log_4 n}$ $= n^{\log_4 3}$ nodes.

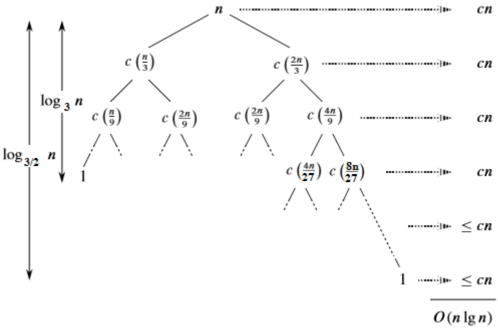
Finally, we add up the costs over all levels to determine the cost of the entire tree.

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \quad \textit{since} \sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \\
\textit{or} &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \quad (\textit{use substitution to verify})
\end{aligned}
$$

$T(n) = T(n/3) + T(2n/3) + O(n)$

T(n) = T(n/3) + T(2n/3) + O(n)

$$\Rightarrow T(n) = T(n/3) + T(2n/3) + cn$$



The longest path from the root to a leaf is n $\rightarrow \frac{2}{3}$n $\rightarrow \left(\frac{2}{3}\right)^2$n $\rightarrow \ldots \rightarrow$ 1. Hence, the height k of the tree is given from: $\left(\frac{2}{3}\right)^k$n $= 1 \Rightarrow$ k $= \log_{3/2} n$.

$$\Rightarrow T(n) < cnlog_{3/2}n$$

$$= O(nlgn)$$

The Master Method

Let $a \geq 1$ and $b > 1$ be constants, and let f(n) is asymptotically positive. Let T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

Then T(n) can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then
$$T(n) = \Theta(n^{log_b a})$$
2. If $f(n) = \Theta(n^{\log_b a} lg^k n)$ for some $k \geq 0$, then
$$T(n) = \Theta(n^{\log_b a} lg^{k+1} n)$$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ (known as the regularity condition) for some constant $c < 1$ and for all sufficiently large n, then
$$T(n) = \Theta(f(n))$$

The Master Method: Interpretation

Compare f(n) with $n^{\log_b a}$

- If $n^{\log_b a}$ is polynomially larger than f(n), use case 1
- If f(n) is polynomially larger than $n^{\log_b a}$ and the regularity condition is satisfied, use case 3
- If the two functions grow at similar rates i.e. f(n) $= \Theta(n^{\log_b a} lg^{k+1} n)$, use case 2

The Master Method: Interpretation

Compare f(n) with $n^{\log_b a}$

- If $n^{\log_b a}$ is polynomially larger than f(n), use case 1
- If f(n) is polynomially larger than $n^{\log_b a}$ and the regularity condition is satisfied, use case 3
- If the two functions grow at similar rates i.e. $\mathrm{f(n)} = \Theta(n^{\log_b a} lg^{k+1} n)$, use case 2

Example: $\mathrm{T(n) = 4T(n/2) + n}$

The Master Method: Interpretation

Compare f(n) with $n^{\log_b a}$

- If $n^{\log_b a}$ is polynomially larger than f(n), use case 1
- If f(n) is polynomially larger than $n^{\log_b a}$ and the regularity condition is satisfied, use case 3
- If the two functions grow at similar rates i.e. f(n) $= \Theta(n^{\log_b a} lg^{k+1} n)$, use case 2

Example: T(n) = 4T(n/2) + n
a = 4, b = 2, f(n) = n
$\Rightarrow n^{\log_2 4} = n^2 \Rightarrow$ f(n) = O($n^{2-\epsilon}$), for $\epsilon = 0.5$
$\Rightarrow$ **Case 1:** T(n) = $\Theta(n^2)$

Examples

1. $T(n) = 4T(n/2) + n^2$

Examples

1. $T(n) = 4T(n/2) + n^2$
   $\mathrm{f(n)} = n^2 = \Theta(n^{\log_2 4}) = \Theta(n^2)$
   $\Rightarrow$ case 2, for k = 0 $\Rightarrow$ T(n) = $\Theta(n^2 lgn)$
2. $T(n) = 4T(n/2) + n^3$

Examples

1. $T(n) = 4T(n/2) + n^2$
   f(n) $= n^2 = \Theta(n^{\log_2 4}) = \Theta(n^2)$
   $\Rightarrow$ case 2, for k $= 0 \Rightarrow$ T(n) $= \Theta(n^2 lgn)$

2. $T(n) = 4T(n/2) + n^3$
   f(n) $= n^3 = \Omega(n^{\log_2 4+0.5})$ for $\epsilon = 0.5 \Rightarrow$ Case 3
   Check regularity condition: $4(n/2)^3 \leq 0.5n^3$ for $c = 0.5$
   $\Rightarrow$ T(n) $= \Theta(n^3)$

3. $T(n) = 3T(n/4) + nlgn$

Examples

1. $T(n) = 4T(n/2) + n^2$
   f(n) $= n^2 = \Theta(n^{\log_2 4}) = \Theta(n^2)$
   $\Rightarrow$ case 2, for k $= 0 \Rightarrow$ T(n) $= \Theta(n^2 lgn)$

2. $T(n) = 4T(n/2) + n^3$
   f(n) $= n^3 = \Omega(n^{\log_2 4+0.5})$ for $\epsilon = 0.5 \Rightarrow$ Case 3
   Check regularity condition: $4(n/2)^3 \leq 0.5n^3$ for $c = 0.5$
   $\Rightarrow$ T(n) $= \Theta(n^3)$

3. $T(n) = 3T(n/4) + nlgn$
   a $=$ 3, b $=$ 4, f(n) $=$ nlgn, and $n^{\log_4 3} = n^{0.793}$
   $\Rightarrow$ f(n) $= \Omega(n^{0.999})$ where $\epsilon \approx 0.2$
   Check regularity condition: 3f(n/4) $=$ 3(n/4)lg(n/4) $\leq$ 3/4nlgn $=$ 0.75nlgn
   $\Rightarrow$ Case 3, where c $=$ 3/4
   $\Rightarrow$ T(n) $= \Theta(nlgn)$

Examples

4 $T(n) = 2T(n/2) + nlgn$

Examples

4. $T(n) = 2T(n/2) + nlgn$
   a = 2, b = 2, f(n) = nlgn and $n^{\log_b a} = n$
   f(n) = $\Theta(n^{\log_b a} lg^k n) = \Theta(nlgn)$ for k = 1
   $\Rightarrow$ Case 2 $\Rightarrow$ T(n) = $\Theta(nlg^2 n)$

5. T(n) = 2T(n/2) + n/lgn

Examples

4 $T(n) = 2T(n/2) + nlgn$
a = 2, b = 2, f(n) = nlgn and $n^{\log_b a} = n$
f(n) = $\Theta(n^{\log_b a} lg^k n) = \Theta(nlgn)$ for k = 1
$\Rightarrow$ Case 2 $\Rightarrow$ T(n) = $\Theta(nlg^2 n)$

5 T(n) = 2T(n/2) + n/lgn
Master method doesn't apply because non-polynomial difference between f(n) and $n^{\log_b a}$

6 T(n) = 0.5T(n/2) + 1/n

Examples

4 $T(n) = 2T(n/2) + nlgn$
a = 2, b = 2, f(n) = nlgn and $n^{\log_b a} = n$
f(n) = $\Theta(n^{\log_b a} lg^k n) = \Theta(nlgn)$ for k = 1
$\Rightarrow$ Case 2 $\Rightarrow$ T(n) = $\Theta(nlg^2 n)$

5 T(n) = 2T(n/2) + n/lgn
Master method doesn't apply because non-polynomial difference between f(n) and $n^{\log_b a}$

6 T(n) = 0.5T(n/2) + 1/n
Master theorem doesn't apply because $a < 1$

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Let $m = lgn$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Let $m = lgn$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.
Now let $S(m) = T(2^m)$.

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Let $m = lgn$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.
Now let $S(m) = T(2^m)$.
$\Rightarrow S(m) = 2S(m/2) + m$.

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Let $m = lgn$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.
Now let $S(m) = T(2^m)$.
$\Rightarrow S(m) = 2S(m/2) + m$.
$\Rightarrow S(m) = O(mlgm)$.

Change of variables

$T(n) = 2T(\sqrt{n}) + lgn$

Let $m = lgn$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.
Now let $S(m) = T(2^m)$.
$\Rightarrow S(m) = 2S(m/2) + m$.
$\Rightarrow S(m) = O(mlgm)$.

$\therefore T(n) = T(2^m) = S(m) = O(mlgm) = O(lgnlglgn)$.

Divid-and-Conquer Algorithms

- The divide-and-conquer paradigm involves three steps at each level of the recursion:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem

The Exponentiation Function: Example

- Suppose we want to find x^n for some nonnegative integer n. The naive way to do it is using a for loop:

Nonrecursive Form

```
Power(base, n)
   answer = 1
   for i = 1 to n
      answer = answer * base
   return answer
```

The Exponentiation Function: Example

- Suppose we want to find x^n for some nonnegative integer n. The naive way to do it is using a for loop:

Nonrecursive Form

```
Power(base, n)
   answer = 1
   for i = 1 to n
      answer = answer * base
   return answer
```

Recursive Form

```
Power(base, n)
  if n==0
     return 1
  else
     return base*Power(base, n-1)
```

The Exponentiation Function: Example

- Suppose we want to find x^n for some nonnegative integer n. The naive way to do it is using a for loop:

Nonrecursive Form

```
Power(base, n)
   answer = 1
   for i = 1 to n
      answer = answer * base
   return answer
```

Recursive Form

```
Power(base, n)
  if n==0
     return 1
  else
     return base*Power(base, n-1)
```

- Both run in $O(n)$ time, since the body of the for loop takes $O(1)$ time to execute, and the for loop test is executed $n+1$ times.
- Can we do better?

Divide and Conquer Approach

We can get a faster algorithm if we can define exponentiation recursively as follows.

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

```
Power(base, n)
  if base = 0 return 1
  else if n mod 2 = 0
     x = Power(base, n/2)
     return x*x
  else
     x = Power(base, (n-1)/2)
     return base*x*x
```

Runtime

$$T(n) = \begin{cases} 1 & \text{if n} = 1, \\ T(n/2) + \Theta(1) & \text{if n is even,} \\ T((n-1)/2) + \Theta(1) & \text{if n is odd.} \end{cases}$$

$$\Rightarrow T(n) = T(n/2) + \Theta(1)$$

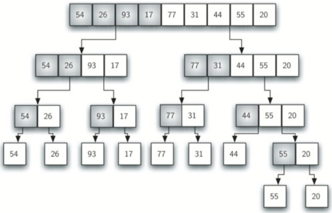
Using case 2 of the Master Method, $T(n) = \Theta(lgn)$.

The Mergesort Algorithm

- Closely follows the divide-and-conquer paradigm.
  - **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
  - **Conquer:** Sort the two subsequences recursively using merge sort.
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer.
- The recursion bottoms out when the sequence to be sorted has length 1, in which case there is no work to be done

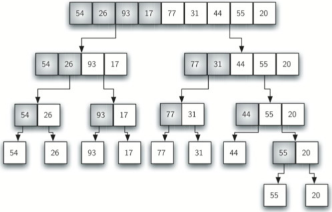
Example

Splitting

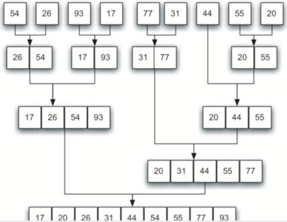


Example

Splitting



Merging



Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
```

if $p < r$ **then**

  $q = \lfloor (p + r)/2 \rfloor$

  `MERGE-SORT(A, p, q)`

  `MERGE-SORT(A, q + 1, r)`

  `MERGE(A, p, q, r)`

```
MERGE(A, p, q, r)
```

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to n_1 **do**
  $L[i] = A[p + i - 1]$
for $j = 1$ to n_2 **do**
  $R[i] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to r **do**
  if $L[i] \leq R[j]$ **then**
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[j]$
    $j = j + 1$

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

```
MERGE-SORT(A,p,r)                    T(n)
1. if p < r
2.       q = ⌊(p + r)/2⌋
3.       MERGE-SORT(A,p,q)
4.       MERGE-SORT(A,q+1,r)
5.       MERGE(A,p,q,r)
```

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $q = \lfloor (p+r)/2 \rfloor$	
3. `MERGE-SORT(A,p,q)`	
4. `MERGE-SORT(A,q+1,r)`	
5. `MERGE(A,p,q,r)`	

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $\quad q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. `MERGE-SORT(A,p,q)`	
4. `MERGE-SORT(A,q+1,r)`	
5. `MERGE(A,p,q,r)`	

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $\quad q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. `MERGE-SORT(A,p,q)`	T(n/2)
4. `MERGE-SORT(A,q+1,r)`	
5. `MERGE(A,p,q,r)`	

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. `MERGE-SORT(A,p,q)`	T(n/2)
4. `MERGE-SORT(A,q+1,r)`	T(n/2)
5. `MERGE(A,p,q,r)`	

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $\quad q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. `MERGE-SORT(A,p,q)`	T(n/2)
4. `MERGE-SORT(A,q+1,r)`	T(n/2)
5. `MERGE(A,p,q,r)`	$\Theta(n)$

Analysis of merge sort

We start with the initial call `MERGE-SORT(A,1,n)`

`MERGE-SORT(A,p,r)`	T(n)
1. **if** $p < r$	$\Theta(1)$
2. $\quad q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. $\quad$ `MERGE-SORT(A,p,q)`	T(n/2)
4. $\quad$ `MERGE-SORT(A,q+1,r)`	T(n/2)
5. $\quad$ `MERGE(A,p,q,r)`	$\Theta(n)$

Therefore,

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

Analysis of Merge sort

Let's rewrite the recurrence as

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

Let's assume that n is the exact power of 2 and we solve the recursion using *recursion tree method*.

T(n)

Analysis of Merge sort

Let's rewrite the recurrence as

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

Let's assume that n is the exact power of 2 and we solve the recursion using *recursion tree method*.

T(n) $\Rightarrow$

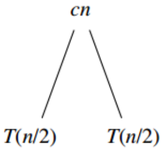
Analysis of Merge sort

Let's rewrite the recurrence as

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

Let's assume that n is the exact power of 2 and we solve the recursion using *recursion tree method*.

T(n) $\Rightarrow$



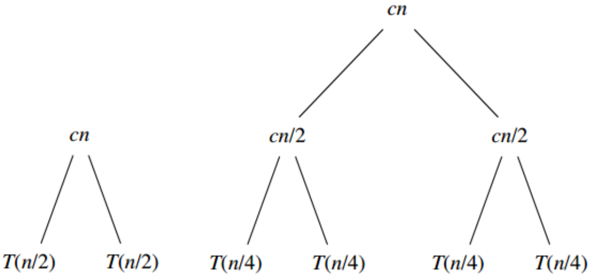
Analysis of Merge sort

Let's rewrite the recurrence as

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

Let's assume that n is the exact power of 2 and we solve the recursion using *recursion tree method*.

$\mathrm{T(n)} \Rightarrow$



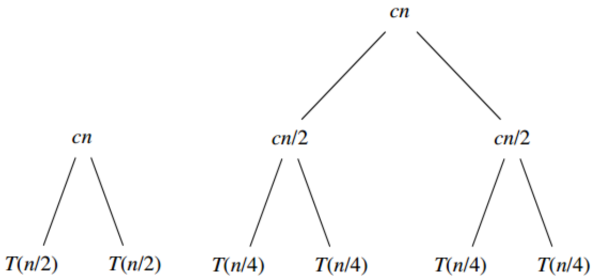
Analysis of Merge sort

Let's rewrite the recurrence as

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

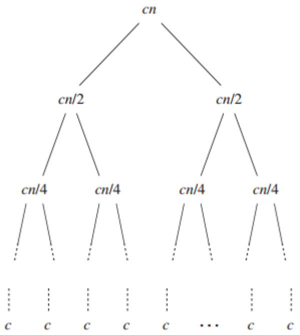
Let's assume that n is the exact power of 2 and we solve the recursion using *recursion tree method*.

T(n) $\Rightarrow$

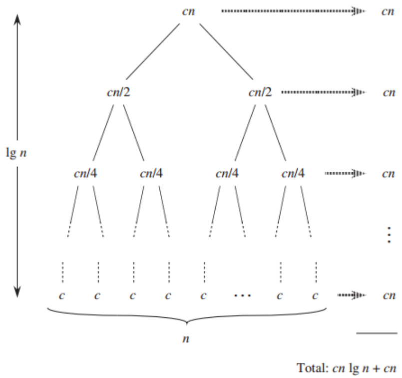


Analysis of Merge sort

We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c.



Analysis of Merge sort

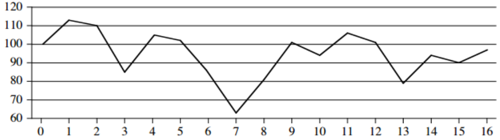


$$\Rightarrow T(n) = \Theta(nlgn)$$

Alternatively, we can use the master method to obtain the same result.

The Maximum Subarray Problem

- Suppose you are given the price of a stock on each day.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- How do you decide when to buy and when to sell to maximize your profit?

- **Naive strategy:** Try all pairs of (buy, sell) dates, where the buy date must be before the sell date. This takes $\Theta(n^2)$.

```
bestProfit = -MAX_INT
bestBuyDate = None
bestSellDate = None
for i = 1 to n
   for j = i + 1 to n
      if price[j] - price[i] > bestProfit
         bestBuyDate = i
         bestSellDate = j
         bestProfit = price[j] - price[i]
```

Divide and Conquer Strategy

- Instead of the daily price, consider the daily change in price, which (on each day) can be either a positive or negative number.
- If we let array A store these changes, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

- We call this contiguous subarray the **maximum subarray**.
- The maximum subarray of `A[1..16]` is `A[8..11]`, with the sum 43.
- So, Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11.

Divide and Conquer Strategy

- Divide the array into two. The maximum subarray must be:
  - Entirely in the first half,
  - Entirely in the second half, or
  - It must span the border between the first and the second half.
- For the first two cases, we can find the solution using a recursive call on a subproblem half as large.
- For the third case, the sum of that array is the sum of two parts.

```
FIND-MAXIMUM-SUBARRAY(A, low , high )
if high == low
  return A[low ] // base case : only one element
else
   mid = ⌊ (low + high)/2 ⌋
   left-sum = FIND-MAXIMUM-SUBARRAY(A, low , mid)
   right-sum = FIND-MAXIMUM-SUBARRAY(A, mid+1, high )
   cross-sum = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
   if left-sum >= right-sum and left-sum >= cross-sum
      return left-sum
   elseif right-sum >= left-sum and right-sum >= cross-sum
      return right-sum
   else
      return cross-sum
```

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -MAX_INT
sum = 0
for i = mid downto low
   sum = sum + A[i]
   if sum > left-sum
      left-sum = sum

right-sum = -MAX_INT
sum = 0
for j = mid + 1 to high
   sum = sum + A[j]
   if sum > right-sum
      right-sum = sum

return left-sum + right-sum
```

Runtime Analysis

- For FindMaxCrossingSubarray: $\Theta(n)$
- For FindMaximumSubArray:
  $\Theta(1) + \Theta(\lfloor n/2 \rfloor) + \Theta(\lceil n/2 \rceil) + \Theta(n) + \Theta(1)$

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

By case 2 of the master theorem, this recurrence has the solution $T(n) = \Theta(nlgn)$.

Exercise

Given array $\texttt{A} = \langle$-2, 1, -3, 4, -1, 2, 1, -5, 4$\rangle$

1. Sort using Mergesort
2. Find the value of the maximum subarray (Use divide and conquer approach).