Reductions, Recursion and Divide and Conquer

Lecture 6

February 5, 2015

Part I

Reductions and Recursion

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a *black box*

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until it turns blue, and then shoot him with the blue elephant gun.

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until it turns blue, and then shoot him with the blue elephant gun.

Q: How do you shoot a white elephant?

A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any *duplicates* in A?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any *duplicates* in A?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any *duplicates* in A?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any *duplicates* in A?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

Reduction to Sorting

1. Code:

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

2. **Running time:** $O(n)$ plus time to sort an array of n numbers
3. **Key point:** algorithm uses sorting as a black box.

Reduction to Sorting

1. Code:

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

2. **Running time:** $O(n)$ plus time to sort an array of n numbers
3. **Key point:** algorithm uses sorting as a black box.

Reduction to Sorting

1. Code:

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

2. **Running time:** $O(n)$ plus time to sort an array of n numbers
3. **Key point:** algorithm uses sorting as a black box.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
   1. Positive direction: Algorithm for B implies an algorithm for A
   2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
   1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
   2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

1. Suppose problem A reduces to problem B
    1. Positive direction: Algorithm for B implies an algorithm for A
    2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
2. **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
    1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
    2. If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

Recursion

1. **Reduction:** reduce one problem to another.
2. **Recursion:** a special case of reduction
   1. reduce problem to a *smaller* instance of *itself*
   2. self-reduction
3. Recursion as a reduction:
   1. Problem instance of size n is reduced to *one or more* instances of size $n - 1$ or less.
   2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. **Reduction:** reduce one problem to another.
2. **Recursion:** a special case of reduction
    1. reduce problem to a *smaller* instance of *itself*
    2. self-reduction
3. Recursion as a reduction:
    1. Problem instance of size n is reduced to *one or more* instances of size $n - 1$ or less.
    2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. **Reduction:** reduce one problem to another.
2. **Recursion:** a special case of reduction
    1. reduce problem to a *smaller* instance of *itself*
    2. self-reduction
3. Recursion as a reduction:
    1. Problem instance of size n is reduced to *one or more* instances of size $n-1$ or less.
    2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. **Reduction:** reduce one problem to another.
2. **Recursion:** a special case of reduction
   1. reduce problem to a *smaller* instance of *itself*
   2. self-reduction
3. Recursion as a reduction:
   1. Problem instance of size n is reduced to *one or more* instances of size $n - 1$ or less.
   2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
   1. Divide and conquer
   2. Dynamic programming
   3. Enumeration and branch and bound etc
   4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
   1. Divide and conquer
   2. Dynamic programming
   3. Enumeration and branch and bound etc
   4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
   1. Divide and conquer
   2. Dynamic programming
   3. Enumeration and branch and bound etc
   4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Recursion

1. Recursion is a powerful and fundamental technique.
2. Basis for several other methods
    1. Divide and conquer
    2. Dynamic programming
    3. Enumeration and branch and bound etc
    4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for SelectSort on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for **SelectSort** on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for SelectSort on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for **SelectSort** on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for **SelectSort** on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for **SelectSort** on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

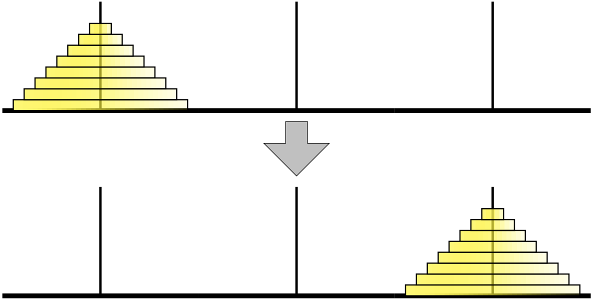
Selection Sort

1. Sort a given array $A[1..n]$ of integers.
2. Recursive version of Selection sort.
3. Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

4. $T(n)$: time for **SelectSort** on an n element array.
5. $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
6. $T(n) = \Theta(n^2)$.

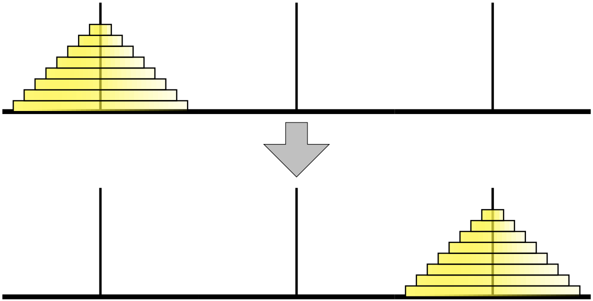
Tower of Hanoi



The Tower of Hanoi puzzle

1. Move stack of n disks from peg **0** to peg **2**, one disk at a time.
2. Rule: cannot put a larger disk on a smaller disk.
3. Question: what is a strategy and how many moves does it take?

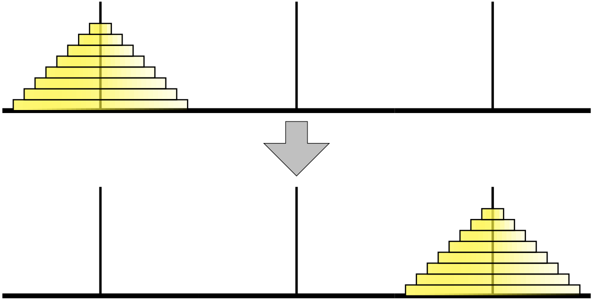
Tower of Hanoi



The Tower of Hanoi puzzle

1. Move stack of n disks from peg **0** to peg **2**, one disk at a time.
2. Rule: cannot put a larger disk on a smaller disk.
3. Question: what is a strategy and how many moves does it take?

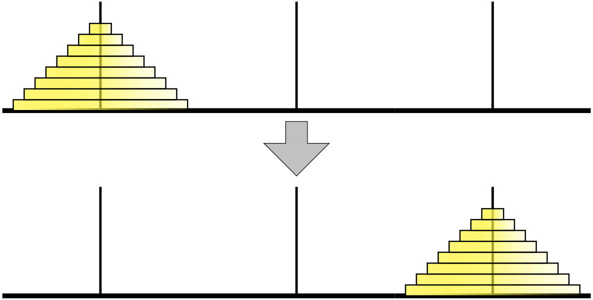
Tower of Hanoi



The Tower of Hanoi puzzle

1. Move stack of n disks from peg **0** to peg **2**, one disk at a time.
2. Rule: cannot put a larger disk on a smaller disk.
3. Question: what is a strategy and how many moves does it take?

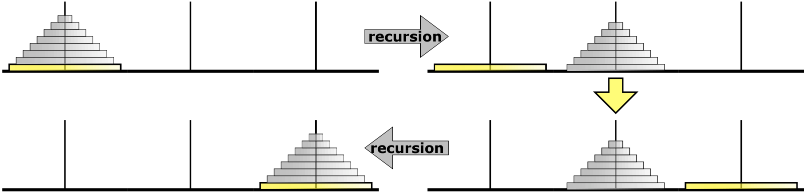
Tower of Hanoi



The Tower of Hanoi puzzle

1. Move stack of n disks from peg $\mathbf{0}$ to peg $\mathbf{2}$, one disk at a time.
2. Rule: cannot put a larger disk on a smaller disk.
3. Question: what is a strategy and how many moves does it take?

Tower of Hanoi via Recursion



The Tower of Hanoi algorithm; ignore everything but the bottom disk

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move n disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move n disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move n disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \dots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \dots + 1 \\
&= \dots \\
&= 2^{n-1} T(1) + 2^{n-2} + \dots + 1 \\
&= 2^{n-1} + 2^{n-2} + \dots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2^2 T(n-2)+2+1 \\
&= \ldots \\
&= 2^i T(n-i)+2^{i-1}+2^{i-2}+\ldots+1 \\
&= \ldots \\
&= 2^{n-1} T(1)+2^{n-2}+\ldots+1 \\
&= 2^{n-1}+2^{n-2}+\ldots+1 \\
&= (2^n-1)/(2-1) = 2^n-1
\end{aligned}
$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Non-Recursive Algorithms for Tower of Hanoi

1. Pegs numbered $0, 1, 2$
2. Non-recursive Algorithm 1:
   1. Always move smallest disk forward if n is even, backward if n is odd.
   2. Never move the same disk twice in a row.
   3. Done when no legal move.
3. Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

1. Pegs numbered $0, 1, 2$
2. Non-recursive Algorithm 1:
    1. Always move smallest disk forward if n is even, backward if n is odd.
    2. Never move the same disk twice in a row.
    3. Done when no legal move.
3. Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

1. Pegs numbered $\mathbf{0, 1, 2}$
2. Non-recursive Algorithm 1:
   1. Always move smallest disk forward if n is even, backward if n is odd.
   2. Never move the same disk twice in a row.
   3. Done when no legal move.
3. Moves are exactly same as those of recursive algorithm. Prove by induction.

Part II

Divide and Conquer

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step
(B) Recursively solve problem on smaller instances.
(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
    1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
    2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step

(B) Recursively solve problem on smaller instances.

(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
   1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
   2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step
(B) Recursively solve problem on smaller instances.
(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
    1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
    2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step
(B) Recursively solve problem on smaller instances.
(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
   1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
   2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step

(B) Recursively solve problem on smaller instances.

(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
    1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
    2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

1. Divide and Conquer is a common and useful type of recursion

Approach

(A) Break problem instance into smaller instances - divide step
(B) Recursively solve problem on smaller instances.
(C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

2. **Question:** Why is this not plain recursion?
    1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
    2. There are many examples of this particular type of recursion that it deserves its own treatment.

Sorting

Input Given an array of n elements

Goal Rearrange them in ascending order

1. Input: Array $A[1 \ldots n]$

$$A \; L \; G \; O \; R \; I \; T \; H \; M \; S$$

Merge Sort [von Neumann]

MergeSort

1. Input: Array $A[1 \ldots n]$

   $A\ L\ G\ O\ R\ I\ T\ H\ M\ S$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $A\ L\ G\ O\ R \quad I\ T\ H\ M\ S$

1. Input: Array $A[1 \ldots n]$

   $A\ L\ G\ O\ R\ I\ T\ H\ M\ S$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $A\ L\ G\ O\ R \quad I\ T\ H\ M\ S$

3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m+1 \ldots n]$

   $A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$

Merge Sort [von Neumann]

MergeSort

1. Input: Array $A[1 \dots n]$

   $$A \; L \; G \; O \; R \; I \; T \; H \; M \; S$$

2. Divide into subarrays $A[1 \dots m]$ and $A[m+1 \dots n]$, where $m = \lfloor n/2 \rfloor$

   $$A \; L \; G \; O \; R \qquad I \; T \; H \; M \; S$$

3. Recursively **MergeSort** $A[1 \dots m]$ and $A[m+1 \dots n]$

   $$A \; G \; L \; O \; R \qquad H \; I \; M \; S \; T$$

4. Merge the sorted arrays

   $$A \; G \; H \; I \; L \; M \; O \; R \; S \; T$$

1. Input: Array $A[1 \ldots n]$

   $$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $$A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$$

3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m+1 \ldots n]$

   $$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

4. Merge the sorted arrays

   $$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

3. Merge two arrays using only constantly more extra space is doable (in-place merge sort).

Merging Sorted Arrays

1. Use a new array C to store the merged array
2. Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

3. Merge two arrays using only constantly more extra space is doable (in-place merge sort).
4. **inplace_merge**: More complicated... Available in STL.

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
    1. $T(n) = O(f(n))$ - upper bound.
    2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
    1. $T(n) = O(f(n))$ - upper bound.
    2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
   1. $T(n) = O(f(n))$ - upper bound.
   2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
    1. $T(n) = O(f(n))$ - upper bound.
    2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
   1. $T(n) = O(f(n))$ - upper bound.
   2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

1. $T(n)$: time for merge sort to sort an n element array
2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
3. What do we want as a solution to the recurrence?
4. Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
   1. $T(n) = O(f(n))$ - upper bound.
   2. $T(n) = \Omega(f(n))$ - lower bound

Solving Recurrences: Some Techniques

1. Some techniques:
    1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
    2. Expand the recurrence and spot a pattern and use simple math.
    3. Recursion tree method — imagine the computation as a tree.
    4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Some techniques:
    1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
    2. Expand the recurrence and spot a pattern and use simple math.
    3. Recursion tree method — imagine the computation as a tree.
    4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Some techniques:
   1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
   2. Expand the recurrence and spot a pattern and use simple math.
   3. Recursion tree method — imagine the computation as a tree.
   4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Some techniques:
   1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
   2. Expand the recurrence and spot a pattern and use simple math.
   3. Recursion tree method — imagine the computation as a tree.
   4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Some techniques:
    1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
    2. Expand the recurrence and spot a pattern and use simple math.
    3. Recursion tree method — imagine the computation as a tree.
    4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Some techniques:
    1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
    2. Expand the recurrence and spot a pattern and use simple math.
    3. Recursion tree method — imagine the computation as a tree.
    4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

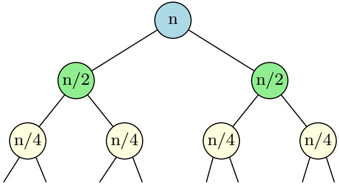
Solving Recurrences: Some Techniques

1. Some techniques:
   1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
   2. Expand the recurrence and spot a pattern and use simple math.
   3. Recursion tree method — imagine the computation as a tree.
   4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
2. **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
3. Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Recursion Trees

MergeSort: **n** is a power of **2**

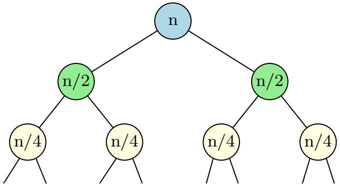
1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$



Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

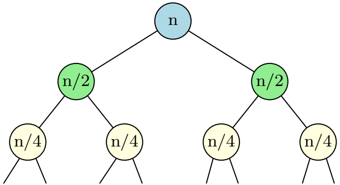


2. Identify a pattern. At the ith level total work is cn.

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

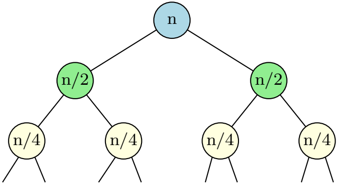


2. Identify a pattern. At the ith level total work is cn.

Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

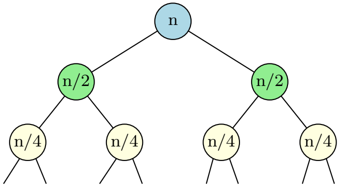


2. Identify a pattern. At the ith level total work is cn.
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.

Recursion Trees

MergeSort: n is a power of 2

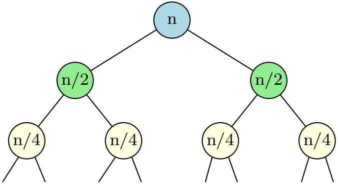
1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$



2. Identify a pattern. At the ith level total work is cn.
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.

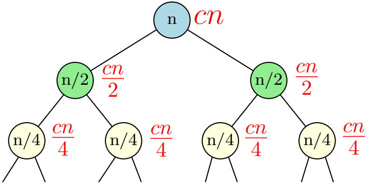
Recursion Trees

An illustrated example...



Recursion Trees

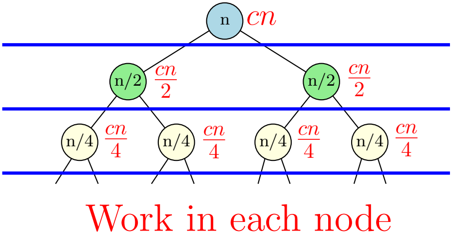
An illustrated example...



Work in each node

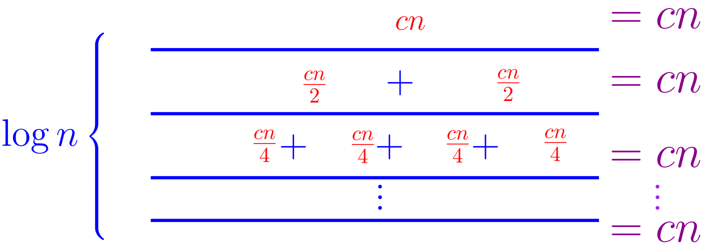
Recursion Trees

An illustrated example...



Recursion Trees

An illustrated example...



Recursion Trees

An illustrated example...

$\log n$ levels:

$$cn = cn$$

$$\frac{cn}{2} + \frac{cn}{2} = cn$$

$$\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn$$

$$\vdots$$

$$= cn$$

$$cn + cn + \cdots + cn$$

$$= cn \log n = O(n \log n)$$

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).
3. $T(n_1) < T(n) \leq T(n_2)$ (Why?).

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).
3. $T(n_1) < T(n) \leq T(n_2)$ (Why?).
4. $T(n) = \Theta(n \log n)$ since $n/2 \leq n_1 < n \leq n_2 \leq 2n$.

Recursion Trees

MergeSort: **n** is not a power of 2

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

Observation: For any number x, $\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$.

MergeSort Analysis

When **n** is not a power of **2**: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

1. If n is power of $\mathbf{2}$ we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

1. If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
2. Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
3. *Verify?*
4. proof by induction!
5. **Induction Hypothesis:** $T(n) \leq 2cn \log n$ for all $n \geq 1$
6. **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
7. **Induction Step** Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

Induction Step

We have

$$
\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\
&\leq 2c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \quad \text{(by indu} \\
&\leq 2c\lfloor n/2 \rfloor \log \lceil n/2 \rceil + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \\
&\leq 2c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log (2n/3) + cn \quad \text{(since } \lceil n/2 \rceil \leq 2n/3 \text{ for all } n \geq 2 \\
&\leq 2cn \log n + cn(1 - 2 \log 3/2) \\
&\leq 2cn \log n + cn(\log 2 - \log 9/4) \\
&\leq 2cn \log n
\end{aligned}
$$

Guess and Verify

The math worked out like magic!
Why was $2cn \log n$ chosen instead of say $4cn \log n$?

1. Do not know upfront what constant to choose.
2. Instead assume that $T(n) \leq \alpha cn \log n$ for some constant α. α will be fixed later.
3. Need to prove that for α large enough the algebra succeeds.
4. In our case... need α such that $\alpha \log 3/2 > 1$.
5. Typically, do the algebra with α and then show that it works... ... if α is chosen to be sufficiently large constant.

How do we know which function to guess?
We don't so we try several "reasonable" functions. With practice and experience we get better at guessing the right function.

Guess and Verify

The math worked out like magic!
Why was $2cn \log n$ chosen instead of say $4cn \log n$?

1. Do not know upfront what constant to choose.
2. Instead assume that $T(n) \leq \alpha cn \log n$ for some constant α. α will be fixed later.
3. Need to prove that for α large enough the algebra succeeds.
4. In our case... need α such that $\alpha \log 3/2 > 1$.
5. Typically, do the algebra with α and then show that it works... ... if α is chosen to be sufficiently large constant.

How do we know which function to guess?
We don't so we try several "reasonable" functions. With practice and experience we get better at guessing the right function.

Guess and Verify

What happens if the guess is wrong?

1. Guessed that the solution to the **MergeSort** recurrence is $T(n) = O(n)$.
2. Try to prove by induction that $T(n) \leq \alpha cn$ for some const' α.
3. **Induction Step:** attempt

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\ &\leq \alpha c \lfloor n/2 \rfloor + \alpha c \lceil n/2 \rceil + cn \\ &\leq \alpha cn + cn \\ &\leq (\alpha + 1)cn \end{aligned}$$

   But need to show that $T(n) \leq \alpha cn$!

4. So guess does not work for any constant α. Suggests that our guess is incorrect.

Guess and Verify

What happens if the guess is wrong?

1. Guessed that the solution to the **MergeSort** recurrence is $T(n) = O(n)$.
2. Try to prove by induction that $T(n) \leq \alpha cn$ for some const' α.
3. **Induction Step:** attempt

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\ &\leq \alpha c \lfloor n/2 \rfloor + \alpha c \lceil n/2 \rceil + cn \\ &\leq \alpha cn + cn \\ &\leq (\alpha + 1) cn \end{aligned}$$

But need to show that $T(n) \leq \alpha cn$!

4. So guess does not work for any constant α. Suggests that our guess is incorrect.

Guess and Verify

What happens if the guess is wrong?

1. Guessed that the solution to the **MergeSort** recurrence is $T(n) = O(n)$.
2. Try to prove by induction that $T(n) \leq \alpha cn$ for some const' α.
3. **Induction Step:** attempt

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\ &\leq \alpha c \lfloor n/2 \rfloor + \alpha c \lceil n/2 \rceil + cn \\ &\leq \alpha cn + cn \\ &\leq (\alpha + 1)cn \end{aligned}$$

But need to show that $T(n) \leq \alpha cn$!

4. So guess does not work for any constant α. Suggests that our guess is incorrect.

Guess and Verify

What happens if the guess is wrong?

1. Guessed that the solution to the **MergeSort** recurrence is $T(n) = O(n)$.
2. Try to prove by induction that $T(n) \leq \alpha cn$ for some const' α.
3. **Induction Step:** attempt

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\ &\leq \alpha c \lfloor n/2 \rfloor + \alpha c \lceil n/2 \rceil + cn \\ &\leq \alpha cn + cn \\ &\leq (\alpha + 1)cn \end{aligned}$$

   But need to show that $T(n) \leq \alpha cn$!
4. So guess does not work for any constant α. Suggests that our guess is incorrect.

Selection Sort vs Merge Sort

1. Selection Sort spends $O(n)$ work to reduce problem from n to $n - 1$ leading to $O(n^2)$ running time.
2. Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$
3. **Question:** Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say k arrays of size n/k each?

Selection Sort vs Merge Sort

1. Selection Sort spends $O(n)$ work to reduce problem from n to $n - 1$ leading to $O(n^2)$ running time.
2. Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$
3. **Question:** Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say k arrays of size n/k each?

Selection Sort vs Merge Sort

1. Selection Sort spends $O(n)$ work to reduce problem from n to $n-1$ leading to $O(n^2)$ running time.
2. Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$
3. **Question:** Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say k arrays of size n/k each?

Quick Sort

1. **QuickSort** [Hoare]:
   1. Pick a pivot element from array
   2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
   3. Linear scan of array it. Time is $O(n)$.
   4. Recursively sort the subarrays, and concatenate them.
2. Example:
   1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
   2. pivot: 16
   3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
   4. put them together with pivot in middle

Quick Sort

1. **QuickSort** [Hoare]:
    1. Pick a pivot element from array
    2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
    3. Linear scan of array it. Time is $O(n)$.
    4. Recursively sort the subarrays, and concatenate them.
2. Example:
    1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
    2. pivot: 16
    3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
    4. put them together with pivot in middle

Quick Sort

1. **QuickSort** [Hoare]:
    1. Pick a pivot element from array
    2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
    3. Linear scan of array it. Time is $O(n)$.
    4. Recursively sort the subarrays, and concatenate them.
2. Example:
    1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
    2. pivot: 16
    3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
    4. put them together with pivot in middle

Quick Sort

1. **QuickSort** [Hoare]:
   1. Pick a pivot element from array
   2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
   3. Linear scan of array it. Time is $O(n)$.
   4. Recursively sort the subarrays, and concatenate them.
2. Example:
   1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
   2. pivot: 16
   3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
   4. put them together with pivot in middle

Time Analysis

1. Let k be the rank of the chosen pivot. Then,
   $T(n) = T(k-1) + T(n-k) + O(n)$

Time Analysis

1. Let k be the rank of the chosen pivot. Then,
$T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) =$
$T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$.
Then, $T(n) = O(n \log n)$.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
    1. Theoretically, median can be found in linear time.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Part III

Fast Multiplication

Multiplying Numbers

Problem Given two n-digit numbers x and y, compute their product.

Grade School Multiplication

Compute "partial product" by multiplying each digit of y with x and adding the partial products.

```
   3141
  ×2718
  -----
  25128
  3141
21987
6282
-------
8537238
```

Time Analysis of Grade School Multiplication

1. Each partial product: $\Theta(n)$
2. Number of partial products: $\Theta(n)$
3. Addition of partial products: $\Theta(n^2)$
4. Total time: $\Theta(n^2)$

A Trick of Gauss

1. Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"
2. Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

3. How many multiplications do we need?
4. Only 3! If we do extra additions and subtractions. Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

A Trick of Gauss

1. Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"
2. Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:
$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$
3. How many multiplications do we need?
4. Only 3! If we do extra additions and subtractions. Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

A Trick of Gauss

1. Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"
2. Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

3. How many multiplications do we need?
4. Only 3! If we do extra additions and subtractions. Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

A Trick of Gauss

1. Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"
2. Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

3. How many multiplications do we need?
4. Only 3! If we do extra additions and subtractions. Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

A Trick of Gauss

1. Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"
2. Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:
   $$(a + bi)(c + di) = ac - bd + (ad + bc)i$$
3. How many multiplications do we need?
4. Only 3! If we do extra additions and subtractions. Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

Divide and Conquer

Assume n is a power of 2 for simplicity and numbers are in decimal.

1. $x = x_{n-1}x_{n-2}\dots x_0$ and $y = y_{n-1}y_{n-2}\dots y_0$
2. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\dots x_{n/2}$ and $x_R = x_{n/2-1}\dots x_0$
3. $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\dots y_{n/2}$ and $y_R = y_{n/2-1}\dots y_0$

Therefore

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Example

$$\begin{aligned} 1234 \times 5678 &= (100 \times 12 + 34) \times (100 \times 56 + 78) \\ &= 10000 \times 12 \times 56 \\ &\quad +100 \times (12 \times 78 + 34 \times 56) \\ &\quad +34 \times 78 \end{aligned}$$

Time Analysis

$$
\begin{aligned}
xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\
&= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R
\end{aligned}
$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$
\begin{aligned}
xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\
&= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R
\end{aligned}
$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2} x_L + x_R)(10^{n/2} y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

State of the Art

Schönhage-Strassen 1971: $O(n \log n \log \log n)$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $O(n \log n 2^{O(\log^* n)})$ time

Conjecture

There is an $O(n \log n)$ time algorithm.

Analyzing the Recurrences

1. Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
2. Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

1. In both cases, depth of recursion $L = \log n$.
2. Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
3. Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Analyzing the Recurrences

1. Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
2. Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

1. In both cases, depth of recursion $L = \log n$.
2. Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
3. Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Recursion tree analysis

Notes

Notes

Notes

Notes