Reductions, Recursion and Divide and Conquer

Lecture 5

February 1, 2011

Part I

Reductions and Recursion

Reduction

Reducing problem **A** to problem **B**:

- Algorithm for **A** uses algorithm for **B** as a *black box*

Distinct Elements Problem

Problem Given an array $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

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: $O(n^2)$

Distinct Elements Problem

Problem Given an array $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

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: $O(n^2)$

Distinct Elements Problem

Problem Given an array $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

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: $O(n^2)$

Distinct Elements Problem

Problem Given an array $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

Naive algorithm:

for $\mathbf{i = 1}$ to $\mathbf{n - 1}$ **do**
  for $\mathbf{j = i + 1}$ to $\mathbf{n}$ **do**
    if ($\mathbf{A[i] = A[j]}$)
      return YES
return NO

Running time: $\mathbf{O(n^2)}$

Reduction to Sorting

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

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Reduction to Sorting

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

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Two sides of Reductions

Suppose problem **A** reduces to problem **B**

- Positive direction: Algorithm for **B** implies an algorithm for **A**
- 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)

Example: Distinct Elements reduces to Sorting in $O(n)$ time

- An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
- 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

Suppose problem **A** reduces to problem **B**

- Positive direction: Algorithm for **B** implies an algorithm for **A**
- 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)

Example: Distinct Elements reduces to Sorting in $O(n)$ time

- An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
- 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

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction
- Problem instance of size $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

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

Selection Sort

Sort a given array $\mathbf{A[1..n]}$ of integers.

Recursive version of Selection sort.

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

$\mathbf{T(n)}$: time for **SelectSort** on an $\mathbf{n}$ element array.

$\mathbf{T(n) = T(n-1) + n}$ for $\mathbf{n > 1}$ and $\mathbf{T(1) = 1}$ for $\mathbf{n = 1}$

$\mathbf{T(n) = \Theta(n^2)}$.

Selection Sort

Sort a given array $\mathbf{A[1..n]}$ of integers.

Recursive version of Selection sort.

```
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])
```

$\mathbf{T(n)}$: time for **SelectSort** on an $\mathbf{n}$ element array.

$\mathbf{T(n) = T(n-1) + n}$ for $\mathbf{n > 1}$ and $\mathbf{T(1) = 1}$ for $\mathbf{n = 1}$

$\mathbf{T(n)} = \Theta(n^2)$.

Selection Sort

Sort a given array **A[1..n]** of integers.

Recursive version of Selection sort.

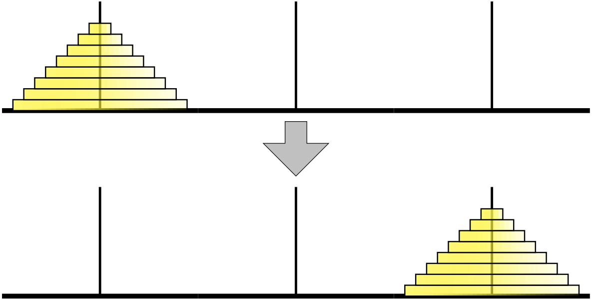
```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.  Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

T(n): time for **SelectSort** on an **n** element array.

$T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$

$T(n) = \Theta(n^2)$.

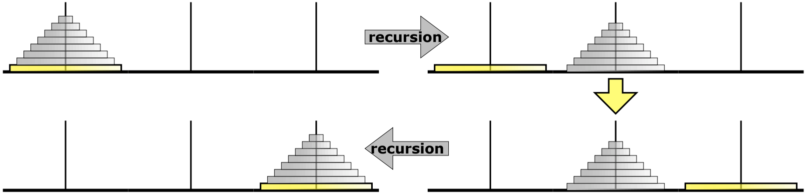
Tower of Hanoi



The Tower of Hanoi puzzle

Move stack of **n** disks from peg **0** to peg **2**, one disk at a time.
Rule: cannot put a larger disk on a smaller disk.
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)
```

$\mathbf{T(n)}$: time to move $\mathbf{n}$ 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)
```

$\mathbf{T(n)}$: time to move $\mathbf{n}$ disks via recursive strategy

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

Analysis

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

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $\mathbf{0, 1, 2}$

Non-recursive Algorithm 1:

- Always move smallest disk forward if $\mathbf{n}$ is even, backward if $\mathbf{n}$ is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

Non-recursive Algorithm 2:

- Let $\rho(\mathbf{n})$ be the smallest integer $\mathbf{k}$ such that $\mathbf{n}/\mathbf{2^k}$ is *not* an integer. Example: $\rho(40) = 4$, $\rho(18) = 2$.
- In step $\mathbf{i}$ move disk $\rho(\mathbf{i})$ forward if $\mathbf{n} - \mathbf{i}$ is even and backward if $\mathbf{n} - \mathbf{i}$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $0, 1, 2$

Non-recursive Algorithm 1:

- Always move smallest disk forward if n is even, backward if n is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

Non-recursive Algorithm 2:

- Let $\rho(n)$ be the smallest integer k such that $n/2^k$ is *not* an integer. Example: $\rho(40) = 4$, $\rho(18) = 2$.
- In step i move disk $\rho(i)$ forward if $n - i$ is even and backward if $n - i$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $0, 1, 2$

Non-recursive Algorithm 1:

- Always move smallest disk forward if n is even, backward if n is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

Non-recursive Algorithm 2:

- Let $\rho(n)$ be the smallest integer k such that $n/2^k$ is *not* an integer. Example: $\rho(40) = 4$, $\rho(18) = 2$.
- In step i move disk $\rho(i)$ forward if $n - i$ is even and backward if $n - i$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Part II

Divide and Conquer

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

- 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.
- There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

- 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.
- There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

- 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.
- There are many examples of this particular type of recursion that it deserves its own treatment.

Sorting

Input Given an array of $\mathbf{n}$ elements

Goal Rearrange them in ascending order

Merge Sort [von Neumann]

MergeSort

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

   A L G O R I T H M S

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

   A L G O R   I T H M S

3. Recursively MergeSort $\mathbf{A}[1 \ldots \mathbf{m}]$ and $\mathbf{A}[\mathbf{m}+1 \ldots \mathbf{n}]$

   A G L O R   H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merge Sort [von Neumann]

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

   A L G O R I T H M S

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

   A L G O R   I T H M S

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

   A G L O R   H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

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

   A L G O R I T H M S

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

   A L G O R   I T H M S

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

   A G L O R   H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merge Sort [von Neumann]

MergeSort

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

   A L G O R I T H M S

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

   A L G O R   I T H M S

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

   A G L O R   H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

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

   A L G O R I T H M S

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

   A L G O R   I T H M S

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

   A G L O R   H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merging Sorted Arrays

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

A G L O R   **H I M S T**

A G H I L M O R S T

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Merging Sorted Arrays

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

A G L O R **H I M S T**

A G H I L M O R S T

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Merging Sorted Arrays

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

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

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Merging Sorted Arrays

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

A G L O R   H I M S T

A G H I L M O R S T

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Merging Sorted Arrays

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

A G L O R   H I M S T

A G H I L M O R S T

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Merging Sorted Arrays

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

A G L O R   H I M S T

A G H I L M O R S T

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

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

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $\mathbf{f(n)}$ such that $\mathbf{T(n) = \Theta(f(n))}$.

- $\mathbf{T(n) = O(f(n))}$ - upper bound
- $\mathbf{T(n) = \Omega(f(n))}$ - lower bound

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

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

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.

- $T(n) = O(f(n))$ - upper bound
- $T(n) = \Omega(f(n))$ - lower bound

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

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

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $\mathbf{f(n)}$ such that $\mathbf{T(n) = \Theta(f(n))}$.

- $\mathbf{T(n) = O(f(n))}$ - upper bound
- $\mathbf{T(n) = \Omega(f(n))}$ - lower bound

Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- Recursion tree method — imagine the computation as a tree
- Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

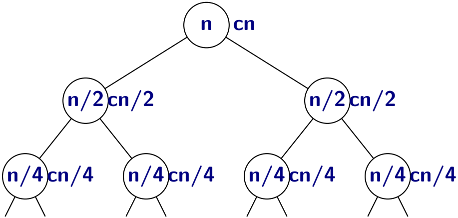
Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- Recursion tree method — imagine the computation as a tree
- Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

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

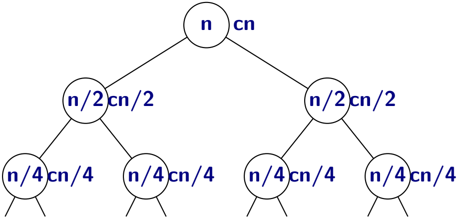


2. Identify a pattern. At the ith level total work is cn
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$

Recursion Trees

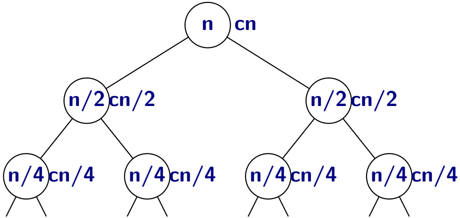
MergeSort: $\mathbf{n}$ is a power of $\mathbf{2}$

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$



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

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

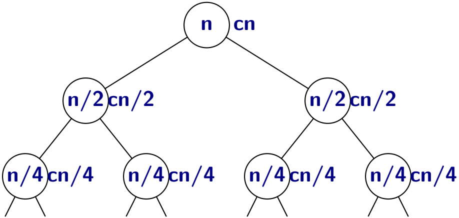


2. Identify a pattern. At the ith level total work is cn
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$

Recursion Trees

MergeSort: n is a power of 2

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

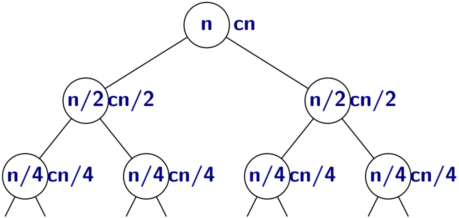


2. Identify a pattern. At the ith level total work is cn
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$

Recursion Trees

MergeSort: $\mathbf{n}$ is a power of $\mathbf{2}$

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$



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

- When n is not a power of 2, the running time of mergesort is expressed as

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

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

MergeSort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

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

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

- When n is not a power of 2, the running time of mergesort is expressed as

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

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

MergeSort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

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

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

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

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

When $\mathbf{n}$ is not a power of $\mathbf{2}$: Guess and Verify

If $\mathbf{n}$ is power of $\mathbf{2}$ we saw that $\mathbf{T(n) = \Theta(n \log n)}$.
Can *guess* that $\mathbf{T(n) = \Theta(n \log n)}$ for all $\mathbf{n}$.
Verify? proof by induction!

Induction Hypothesis: $\mathbf{T(n) \leq 2cn \log n}$ for all $\mathbf{n \geq 1}$
Base Case: $\mathbf{n = 1}$. $\mathbf{T(1) = 0}$ since no need to do any work and $\mathbf{2cn \log n = 0}$ for $\mathbf{n = 1}$.
Induction Step Assume $\mathbf{T(k) \leq 2ck \log k}$ for all $\mathbf{k < n}$ and prove it for $\mathbf{k = n}$.

When $\mathbf{n}$ is not a power of $\mathbf{2}$: Guess and Verify

If $\mathbf{n}$ is power of $\mathbf{2}$ we saw that $\mathbf{T(n) = \Theta(n \log n)}$.
Can *guess* that $\mathbf{T(n) = \Theta(n \log n)}$ for all $\mathbf{n}$.
Verify? proof by induction!

Induction Hypothesis: $\mathbf{T(n) \leq 2cn \log n}$ for all $\mathbf{n \geq 1}$
Base Case: $\mathbf{n = 1}$. $\mathbf{T(1) = 0}$ since no need to do any work and $\mathbf{2cn \log n = 0}$ for $\mathbf{n = 1}$.
Induction Step Assume $\mathbf{T(k) \leq 2ck \log k}$ for all $\mathbf{k < n}$ and prove it for $\mathbf{k = n}$.

Induction Step

We have

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

Guess and Verify

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

Typically we don't know upfront what constant to choose. Instead we assume that $\mathbf{T(n)} \leq \alpha \mathbf{cn \log n}$ for some constant α that will be fixed later. All we need to prove that there is some sufficiently large constant α that will make the algebra go through.

We need to choose α such that $\alpha \log 3/2 > 1$.

Typically you do the algebra with α and then show at the end that α can be chosen to be sufficiently large constant.

Guess and Verify

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

Typically we don't know upfront what constant to choose. Instead we assume that $\mathbf{T(n)} \leq \alpha \mathbf{cn \log n}$ for some constant α that will be fixed later. All we need to prove that there is some sufficiently large constant α that will make the algebra go through.

We need to choose α such that $\alpha \log 3/2 > 1$.

Typically you do the algebra with α and then show at the end that α can be chosen to be sufficiently large constant.

Guess and Verify: When is a guess incorrect?

Suppose we guessed that the soln to the mergesort recurrent is $\mathbf{T(n) = O(n)}$. We try to prove by induction that $\mathbf{T(n) \leq \alpha cn}$ for some constant α.

Induction Step: attempt

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

But we want to show that $\mathbf{T(n) \leq \alpha cn}$! So guess does not work for *any* constant α. Suggests that our guess is incorrect.

Selection Sort vs Merge Sort

- Selection Sort spends $\mathbf{O(n)}$ work to reduce problem from $\mathbf{n}$ to $\mathbf{n-1}$ leading to $\mathbf{O(n^2)}$ running time.
- Merge Sort spends $\mathbf{O(n)}$ time *after* reducing problem to two instances of size $\mathbf{n/2}$ each. Running time is $\mathbf{O(n \log n)}$

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $\mathbf{k}$ arrays of size $\mathbf{n/k}$ each?

Selection Sort vs Merge Sort

- Selection Sort spends $\mathbf{O(n)}$ work to reduce problem from $\mathbf{n}$ to $\mathbf{n-1}$ leading to $\mathbf{O(n^2)}$ running time.
- Merge Sort spends $\mathbf{O(n)}$ time *after* reducing problem to two instances of size $\mathbf{n/2}$ each. Running time is $\mathbf{O(n \log n)}$

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $\mathbf{k}$ arrays of size $\mathbf{n/k}$ each?

Quick Sort[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. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Quick Sort[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. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Quick Sort

Quick Sort[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. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Quick Sort[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. Linear scan of array does it. Time is $\mathbf{O(n)}$
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Time Analysis

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

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

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

Time Analysis

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

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

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

Time Analysis

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

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

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

Time Analysis

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

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

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

Part III

Fast Multiplication

Multiplying Numbers

Problem Given two $\mathbf{n}$-digit numbers $\mathbf{x}$ and $\mathbf{y}$, compute their product.

Grade School Multiplication

Compute "partial product" by multiplying each digit of $\mathbf{y}$ with $\mathbf{x}$ and adding the partial products.

$$
\begin{array}{r}
3141 \\
\times 2718 \\
\hline
25128 \\
3141 \\
21987 \\
6282 \\
\hline
8537238
\end{array}
$$

Time Analysis of Grade School Multiplication

- Each partial product: $\mathbf{\Theta(n)}$
- Number of partial products: $\mathbf{\Theta(n)}$
- Addition of partial products: $\mathbf{\Theta(n^2)}$
- Total time: $\mathbf{\Theta(n^2)}$

A Trick of Gauss

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

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

How many multiplications do we need?

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

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a+bi)}$ and $\mathbf{(c+di)}$

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

How many multiplications do we need?

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

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

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

How many multiplications do we need?

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 $\mathbf{n}$ is a power of $\mathbf{2}$ for simplicity and numbers are in decimal.

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

Therefore

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

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

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

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

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

$\mathbf{4}$ recursive multiplications of number of size $\mathbf{n/2}$ 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

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

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

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

$\mathbf{4}$ recursive multiplications of number of size $\mathbf{n/2}$ 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

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

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

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

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

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

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

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

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: $\mathbf{O(n \log n \log \log n)}$ time using Fast-Fourier-Transform (FFT)

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

Conjecture: there is an $\mathbf{O(n \log n)}$ time algorithm

Analyzing the Recurrences

- Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
- 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:

- In both cases, depth of recursion $L = \log n$.
- 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
- Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Analyzing the Recurrences

- Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
- 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:

- In both cases, depth of recursion $L = \log n$.
- 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
- 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