CS 577 - Divide and Conquer

Marc Renault

Department of Computer Sciences
University of Wisconsin – Madison

Spring 2021
TopHat Join Code: 524741

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Divide and Conquer

Divide and Conquer (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Divide and Conquer (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Tendencies of DC

- Naturally recursive solutions
- Solving complexities often involve recurrences.
- Often used to improve efficiency of efficient solutions, e.g. $O(n^2) \to O(n \log n)$.
- Used in conjunction with other techniques.

Searching

Linear Search

- Brute force approach: check every item in order.
- TopHat 1: What is the time complexity to search through n items?

Searching

Linear Search

- Brute force approach: check every item in order.
- Time complexity: $O(n)$

Searching

Linear Search

- Brute force approach: check every item in order.
- Time complexity: $O(n)$

Divide and Conquer Approach

Searching

Linear Search

- Brute force approach: check every item in order.
- Time complexity: $O(n)$

Divide and Conquer Approach

- Binary Search

Searching

Linear Search

- Brute force approach: check every item in order.
- Time complexity: $O(n)$

Divide and Conquer Approach

- Binary Search
- Complexity: $O(\log n)$

MergeSort

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- TopHat 2: What is the time complexity?

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- TopHat 3: What is the time complexity?

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $(O(n^2))$, Merge Sort

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $\left(O(n^2)\right)$, Merge Sort
- TopHat 4: What is the time complexity of Merge Sort?

Sorting

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $(O(n^2))$, Merge Sort $(O(n \log n))$

Sorting

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $\left(O(n^2)\right)$, Merge Sort $(O(n \log n))$

Trick Sorts

- Radix Sort $\left(O\left(n\lceil \log k \rceil\right)\right)$, Counting Sort $\left(O\left(n+k\right)\right)$
- k is the maximum key size.

Sorting

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $\left(O(n^2)\right)$, Merge Sort $(O(n \log n))$

Trick Sorts

- Radix Sort $(O(n \lceil \log k \rceil))$, Counting Sort $(O(n+k))$
- k is the maximum key size.
- TopHat 5: What value of k would make both sorts have time complexity no better than Merge Sort?

Sorting

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts

- Divide & Conquer: Quick Sort $\left(O(n^2)\right)$, Merge Sort $(O(n \log n))$

Trick Sorts

- Radix Sort $(O(n\lceil \log k \rceil))$, Counting Sort $(O(n+k))$
- k is the maximum key size.
- TopHat 5: What value of k would make both sorts have time complexity no better than Merge Sort? $\Omega(n \log n)$

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.

if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1, A_2)

Algorithm: Merge

Input : Two lists of comparable items: A and B.
Output: A merged list.

Initialize S to an empty list.
while *either A or B is not empty* **do**
    Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S.
end
return S

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MERGESORT(Front-half of A)
$A_2 :=$ MERGESORT(Back-half of A)
return *MERGE*(A_1,A_2)

Algorithm: MERGE

Input : Two lists of comparable items: A and B.
Output: A merged list.
Initialize S to an empty list.
while *either A or B is not empty* **do**
    Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S.
end
return S

TopHat 6: What is the complexity of MERGE?

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1, A_2)

Algorithm: Merge

Input : Two lists of comparable items: A and B.
Output: A merged list.
Initialize S to an empty list.
while *either A or B is not empty* **do**
    Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S.
end
return S

TopHat 6: What is the complexity of Merge? $O(n)$

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1, A_2)

Program Correctness:

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1, A_2)

Program Correctness:

1. Invariant:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MERGESORT(Front-half of A)
$A_2 :=$ MERGESORT(Back-half of A)
return *MERGE*(A_1,A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MERGESORT.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MERGESORT(Front-half of A)
$A_2 :=$ MERGESORT(Back-half of A)
return *MERGE*(A_1, A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MERGESORT.
   Proof:

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MergeSort.
   Proof: By strong induction on list length:

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MergeSort.
   Proof: By strong induction on list length:
   Base cases: $k = 1$: List is sorted; $k = 2$: Split into size 1, then Merge produces sorted list.

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
A_1 := MergeSort(Front-half of A)
A_2 := MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MergeSort.
   Proof: By strong induction on list length:
   Base cases: $k = 1$: List is sorted; $k = 2$: Split into size 1, then Merge produces sorted list.
   Inductive step: By ind hyp, A_1 and A_2 are sorted, and, then, by definition, Merge will produce a sorted list.

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1, A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MergeSort.
2. Soundness: Immediate from invariant.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MERGESORT(Front-half of A)
$A_2 :=$ MERGESORT(Back-half of A)
return *MERGE*(A_1,A_2)

Program Correctness:

1. Invariant: List A is sorted after call to MERGESORT.
2. Soundness: Immediate from invariant.
3. Complete: Handles lists of any size, and each recursion makes progress towards base case by splitting the list in half.

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return $\textit{Merge}(A_1, A_2)$

Run time Considerations:

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
$A_1 :=$ MergeSort(Front-half of A)
$A_2 :=$ MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

Run time Considerations:

- Cost to Merge: $O(n)$.

MergeSort

Algorithm: MergeSort

Input : A list A of n comparable items.
Output: A sorted list A.
if $|A| = 1$ **then return** A
A_1 := MergeSort(Front-half of A)
A_2 := MergeSort(Back-half of A)
return *Merge*(A_1,A_2)

Run time Considerations:

- Cost to Merge: $O(n)$.
- Recurrences: 2 calls to MergeSort with lists half the size.

MergeSort Recurrence

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

MergeSort Recurrence

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Notes

- More precise: $T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn$
- Usually, we can asymptotically ignore floor and ceilings.
- Essentially, we are assuming n is a power of 2.
- Alternate form: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n); T(1) \leq O(1)$

MergeSort Recurrence

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Notes

- More precise: $T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn$
- Usually, we can asymptotically ignore floor and ceilings.
- Essentially, we are assuming n is a power of 2.
- Alternate form: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n); T(1) \leq O(1)$

Methods

- Unwind / Recurrence Tree
- Guess
- Master Theorem
- Nuclear Bomb Theorem / Master Master Theorem

Unwind MergeSort Recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

Unwind MergeSort Recurrence

$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn
\end{aligned}
$$

Unwind MergeSort Recurrence

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

Unwind MergeSort Recurrence

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

Unwind MergeSort Recurrence

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

$$
\begin{aligned}
& 1 = \frac{n}{2^k} \\
\iff\ & 2^k = n \\
\iff\ & k = \log_2(n)
\end{aligned}
$$

Unwind MergeSort Recurrence

$$
\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\le 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\le 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n)
\end{aligned}
$$

$$
\begin{aligned}
1 &= \frac{n}{2^k} \\
\iff 2^k &= n \\
\iff k &= \log_2(n)
\end{aligned}
$$

Unwind MergeSort Recurrence

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n) \\
&= cn + cn\log n
\end{aligned}$$

$$\begin{aligned}
1 &= \frac{n}{2^k} \\
\iff 2^k &= n \\
\iff k &= \log_2(n)
\end{aligned}$$

Unwind MergeSort Recurrence

$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n) \\
&= cn + cn\log n \\
&= O(n\log(n))
\end{aligned}
$$

$$
\begin{aligned}
1 &= \frac{n}{2^k} \\
\iff 2^k &= n \\
\iff k &= \log_2(n)
\end{aligned}
$$

Recursion Tree Method

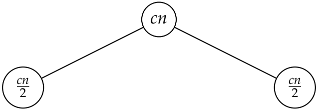
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

cn

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

cn

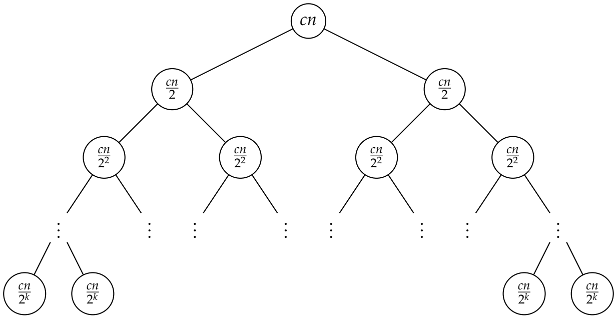
$\frac{cn}{2}$ $\frac{cn}{2}$

$\frac{cn}{2^2}$ $\frac{cn}{2^2}$ $\frac{cn}{2^2}$ $\frac{cn}{2^2}$

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

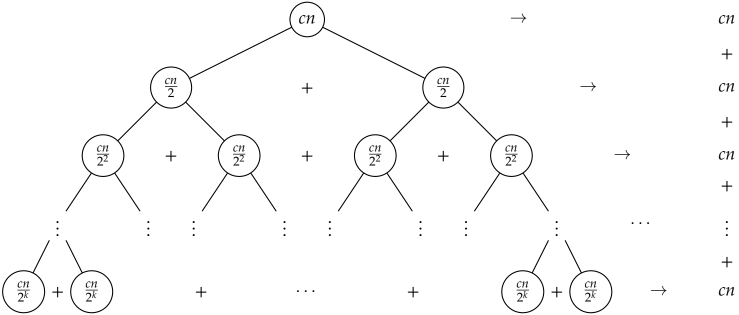
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

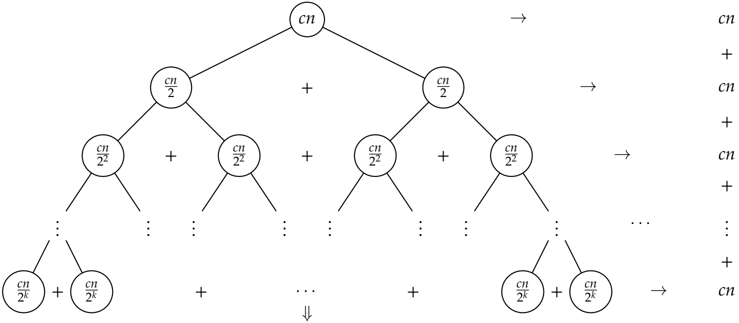


=

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



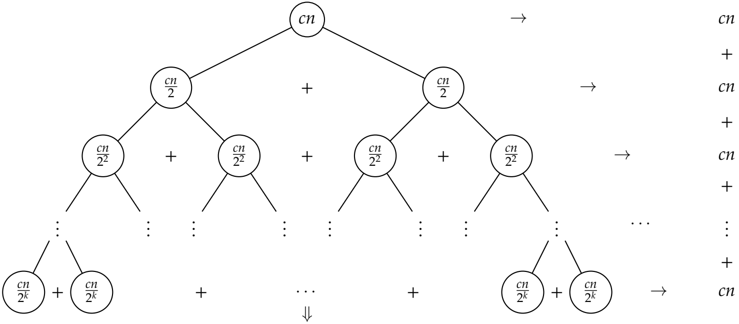
$$\Downarrow$$

$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) =$$

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

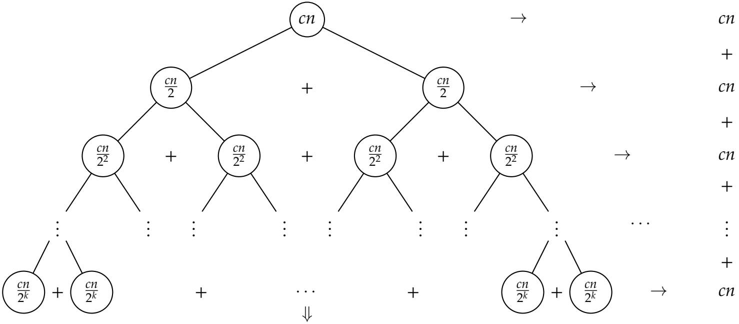


$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) = (k+1)cn = (\text{tree height} + 1) \cdot cn$$

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Recursion Tree Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) = (k+1)cn = (\text{tree height} + 1) \cdot cn \quad = cn \lg n + cn$$
$$= O(n \cdot \lg n)$$

[1] Based on: `http://www.texample.net/tikz/examples/merge-sort-recursion-tree/`

Guess Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Procedure

1. Guess: Seems like $O(n \log n)$-ish.

Guess Method

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Procedure

1. Guess: Seems like $O(n \log n)$-ish.
2. Prove by induction! Not valid without proof!

Prove Recurrence by Strong Induction

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Prove Recurrence by Strong Induction

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) = 2 \cdot T(1) + 2c &\leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

Prove Recurrence by Strong Induction

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) = 2 \cdot T(1) + 2c &\leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

Inductive step:

$$\begin{aligned} T(k) &= 2 \cdot T(k/2) + ck \\ &\leq 2\left(\frac{ck}{2} \lg \frac{k}{2} + \frac{ck}{2}\right) + ck \\ &= ck \lg(k/2) + 2ck \\ &= ck \lg k - ck + 2ck \\ &= ck \lg k + ck \end{aligned}$$

Prove Recurrence by Strong Induction

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) = 2 \cdot T(1) + 2c &\leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

Inductive step:

$$\begin{aligned} T(k) &= 2 \cdot T(k/2) + ck \\ &\leq 2\left(\frac{ck}{2} \lg \frac{k}{2} + \frac{ck}{2}\right) + ck \\ &= ck \lg(k/2) + 2ck \\ &= ck \lg k - ck + 2ck \\ &= ck \lg k + ck \end{aligned}$$

$\therefore O(n \log n)$

Generalized Recurrence

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Generalized Recurrence

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

Generalized Recurrence

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O\left(n^{\lg q}\right)$

Generalized Recurrence

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O\left(n^{\lg q}\right)$

Case $q = 2$

$O\left(n \log n\right)$

Generalized Recurrence

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O\left(n^{\lg q}\right)$

Case $q = 2$

$O\left(n \log n\right)$

Case $q = 1$

Generalized Recurrence

$$T(n) \le q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \le c$$

Case $q > 2$

$O\left(n^{\lg q}\right)$

Case $q = 2$

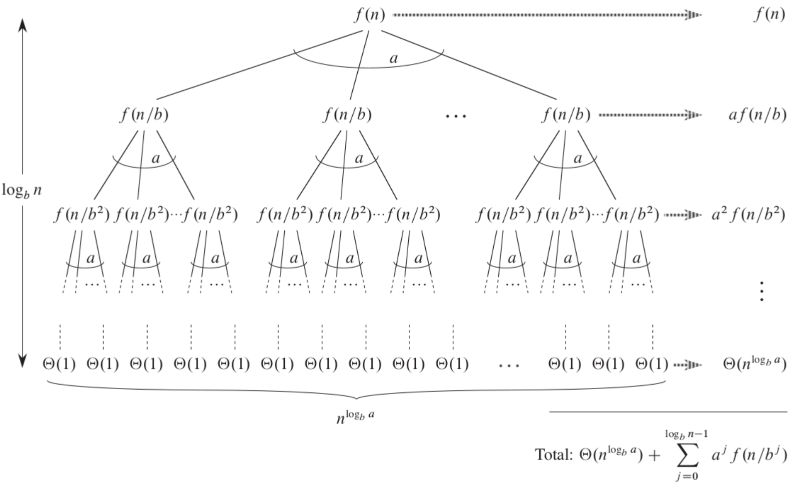
$O\left(n \log n\right)$

Case $q = 1$

$O\left(n\right)$

Master Theorem

Cookbook Recurrence Solving



Master Theorem

Cookbook Recurrence Solving

Theorem 1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

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

where we interpret $n = b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.*
3. *If $\Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Nuclear Bomb / Master Master Theorem

Akra and Bazzi, 1998

Theorem 2

Given a recurrence of the form:

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n) \ ,$$

where k *is a constant,* $a_i > 0$ *and* $b_i > 1$ *are constants for all* i*, and* $f(n) = \Omega(n^c)$ *and* $f(n) = O\left(n^d\right)$ *for some constants* $0 < c \leq d$*. Then,*

$$T(n) = \Theta\left(n^\rho \left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}} du\right)\right) \ ,$$

where ρ *is the unique real solution to the equation*

$$\sum_{i=1}^{k} \frac{a_i}{b_i^\rho} = 1 \ .$$

Inversion Count

Counting Inversions

Inversion

Given a list A of comparable items. An inversion is a pair of items (a_i, a_j) such that $a_i > a_j$ and $i < j$, where i and j are the index of the items in A.

Counting Inversions

Inversion

Given a list A of comparable items. An inversion is a pair of items (a_i, a_j) such that $a_i > a_j$ and $i < j$, where i and j are the index of the items in A.

Inversion Count

Count the number of inversions in a list A, containing n comparable items.

Counting Inversions

Inversion

Given a list A of comparable items. An inversion is a pair of items (a_i, a_j) such that $a_i > a_j$ and $i < j$, where i and j are the index of the items in A.

Inversion Count

Count the number of inversions in a list A, containing n comparable items.

Exercise – Teams of 2 or 3

- Solve the problem in $\Theta(n^2)$.
- Solve the problem in $O(n \log n)$.
- Prove correctness and complexity.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllPairs

Input : A list A of n comparable items.
Output: Number of inversions in A.

```
Let c := 0
for i := 1 to len(A) − 1 do
    for j := i to len(A) do
        if A[i] > A[j] then
            c := c + 1
        end
    end
end
return c
```

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllPairs

Input : A list A of n comparable items.
Output: Number of inversions in A.

Let $c := 0$
for $i := 1$ *to* $len(A) - 1$ **do**
  for $j := i$ *to* $len(A)$ **do**
    if $A[i] > A[j]$ **then**
      $c := c + 1$
    end
  end
end
return c

Analysis

- Correct: Checks all pairs and counts the inversions.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllPairs

Input : A list A of n comparable items.
Output: Number of inversions in A.

```
Let c := 0
for i := 1 to len(A) − 1 do
    for j := i to len(A) do
        if A[i] > A[j] then
            c := c + 1
        end
    end
end
return c
```

Analysis

- Correct: Checks all pairs and counts the inversions.
- Complexity: For each i, check $n - i$ pairs. Overall:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2) \ .$$

Part 2: Give an $O(n \log n)$ solution.

Algorithm: CountSort

Input : A list A of n comparable items.
Output: A sorted array and the number of inversions.
if $|A| = 1$ **then return** $(A, 0)$
$(A_1, c_1) :=$ CountSort(Front-half of A)
$(A_2, c_2) :=$ CountSort(Back-half of A)
$(A, c) :=$MergeCount(A_1,A_2)
return $(A, c + c_1 + c_2)$

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MergeCount

Input : Two lists of comparable items: A and B.
Output: A merged list and the count of inversions.

Initialize S to an empty list and $c := 0$.
while *either A or B is not empty* **do**
  Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S.
  if *Appended item is from B* **then**
    $c := c + |A|$.
  end
end
return (S, c)

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MergeCount

Input : Two lists of comparable items: A and B.
Output: A merged list and the count of inversions.

Initialize S to an empty list and $c := 0$.
while *either A or B is not empty* **do**
  Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S.
  if *Appended item is from B* **then**
    $c := c + |A|$.
  end
end
return (S, c)

Analysis

- Correctness: Need to show that the inversions are counted.
- Complexity: Same recurrence as MergeSort.

Linear Time Selection

Linear Time Selection

Problem

Find the kth value in an unsorted array A of n numbers if A were sorted.

LINEAR TIME SELECTION

Problem

Find the kth value in an unsorted array A of n numbers if A were sorted.

Algorithm: QUICKSELECT

Input : A array $A[1..n]$ and an int k.
Output: The kth element of A if A were sorted.

```
if n = 1 then return A[1]
Choose a pivot A[p]
r := PARTITION(A[1..n], p)
if k < r then
    return QUICKSELECT(A[1..r − 1], k)
else if k > r then
    return QUICKSELECT(A[r + 1..n], k − r)
else
    return A[r]
end
```

Partition around a Pivot

1	2	3	4	5	6	7	8	9
Q	U	I	C	K	S	O	R	T

Pivot: T

$\Downarrow$

1	2	3	4	5	6	7	8	9
Q	I	C	K	S	O	R	T	U

Partition around a Pivot

1	2	3	4	5	6	7	8	9	
Q	U	I	C	K	S	O	R	T	Pivot: Q

$\Downarrow$

1	2	3	4		5		6	7	8	9
I	C	K	O		Q		U	S	R	T

Partition around a Pivot

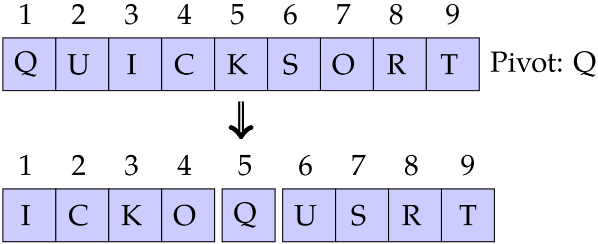
1	2	3	4	5	6	7	8	9	
Q	U	I	C	K	S	O	R	T	Pivot: Q

$\Downarrow$

1	2	3	4	5	6	7	8	9
I	C	K	O	Q	U	S	R	T

TopHat 7: How much work is done to partition around a pivot?

Partition around a Pivot



How much work is done to partition around a pivot? $O(n)$

QuickSelect Recurrence

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + cn$$

Algorithm: QuickSelect

Input : A array $A[1..n]$ and an int k.
Output: The kth element of A.
if $n = 1$ **then return** $A[1]$
Choose a pivot $A[p]$
$r :=$ Partition$(A[1..n], p)$
if $k < r$ **then**
  return *QuickSelect*$(A[1..r-1], k)$
else if $k > r$ **then**
  return *QuickSelect*$(A[r+1..n], k-r)$
else
  return $A[r]$
end

QuickSelect Recurrence

$$
\begin{aligned}
T(n) &\leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + cn \\
&\leq \max_{1 \leq \ell \leq n-1} T(\ell) + cn \\
&\leq T(n-1) + cn \\
&\in O(n^2)
\end{aligned}
$$

Median of Medians

Algorithm: MomSelect

Input : A array $A[1..n]$ and an int k.
Output: The kth element of A.

if *n is small* **then** Solve by brute force.
$m := \lceil n/5 \rceil$
for $i := 1$ *to* m **do**
  $M[i] :=$ brute force find median of $A[5i - 4..5i]$
end
$mom := \text{MomSelect}(M[1..m], \lfloor m/3 \rfloor)$
$r := \text{Partition}(A[1..n], mom)$
if $k < r$ **then**
  return *MomSelect*$(A[1..r-1], k)$
else if $k > r$ **then**
  return *MomSelect*$(A[r+1..n], k-r)$
else
  return $A[r]$
end

MomSelect Analysis

MomSelect Pivot

- greater and less than $> \lfloor \lceil n/5 \rceil /2 \rfloor - 1 \approx n/10$ medians.
- Therefore, MomSelect Pivot is greater and less than $3n/10$ items.
- So, worst-case partition size is $7n/10$.

MomSelect Analysis

MomSelect Pivot

- greater and less than $> \lfloor \lceil n/5 \rceil /2 \rfloor - 1 \approx n/10$ medians.
- Therefore, MomSelect Pivot is greater and less than $3n/10$ items.
- So, worst-case partition size is $7n/10$.

Recurrence:

$$T(n) \leq T(n/5) + T(7n/10) + cn \in O(n)$$

Integer Multiplication

Integer Multiplication

Partial Product Method:

$$
\begin{array}{r}
12 \\
\times\ 13 \\
\hline
36 \\
12 \\
\hline
156
\end{array}
\qquad\qquad
\begin{array}{r}
1100 \\
\times\ 1101 \\
\hline
1100 \\
0000 \\
1100 \\
1100 \\
\hline
10011100
\end{array}
$$

Problem

Multiple two binary numbers x and y, counting every bitwise operation.

Integer Multiplication

Partial Product Method:

$$\begin{array}{r} 12 \\ \times\ 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times\ 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

Problem

Multiple two binary numbers x and y, counting every bitwise operation.

TopHat 8: What is the complexity of the partial product method?

Integer Multiplication

Partial Product Method:

$$\begin{array}{r} 12 \\ \times\ 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times\ 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

Problem

Multiple two binary numbers x and y, counting every bitwise operation.

TopHat 8: What is the complexity of the partial product method? $O(n^2)$.

Divide & Conquer v1

TopHat Discussion 1: Suggest how to divide the problem.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- TH9: How many recursive calls?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call? $O(n)$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call? $O(n)$
- TH10: What is the size of the recursive calls?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$
\begin{aligned}
xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\
&= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0
\end{aligned}
$$

- How many recursive calls? 4.
- Cost per call? $O(n)$
- What is the size of the recursive calls? $n/2$.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call? $O(n)$
- What is the size of the recursive calls? $n/2$.
- TH11: What is the recurrence?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call? $O(n)$
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 4T(n/2) + cn$$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

- How many recursive calls? 4.
- Cost per call? $O(n)$
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 4T(n/2) + cn = O\left(n^{\lg 4}\right) = O\left(n^2\right)$$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0y_0 := \texttt{intMult}(x_0, y_0)$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1 y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0 y_0 := \texttt{intMult}(x_0, y_0)$
- Combine: Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1 y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0 y_0 := \texttt{intMult}(x_0, y_0)$
- Combine: Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$
- Recurrence: $T(n) \leq 3T(n/2) + O(n) = O\left(n^{\lg 3}\right) = O\left(n^{1.59}\right)$

Matrix Multiplication

MATRIX MULTIPLICATION

Problem

Multiple two *nxn* matrices, A and B. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 2 & 1 \cdot 3 + 2 \cdot 1 \\ 3 \cdot 4 + 4 \cdot 2 & 3 \cdot 3 + 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Matrix Multiplication

Problem

Multiple two *nxn* matrices, A and B. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 2 & 1 \cdot 3 + 2 \cdot 1 \\ 3 \cdot 4 + 4 \cdot 2 & 3 \cdot 3 + 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Algorithm: Naïve Method

for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
        for $k \leftarrow 1$ **to** n **do**
            $C[i][j] += A[i][k] \cdot B[k][j]$
        end
    end
end

TopHat 12: What is the complexity of the Naïve Method?

Matrix Multiplication

Problem

Multiple two nxn matrices, A and B. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1\cdot 4 + 2\cdot 2 & 1\cdot 3 + 2\cdot 1 \\ 3\cdot 4 + 4\cdot 2 & 3\cdot 3 + 4\cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Algorithm: Naïve Method

```
for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            C[i][j] += A[i][k] · B[k][j]
        end
    end
end
```

TopHat 12: What is the complexity of the Naïve Method? $O(n^3)$.

Divide & Conquer v1

TopHat Discussion 2: Suggest how to divide the problem.

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- TH13: How many recursive calls?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- TH14: What is the size of the recursive calls?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- TH15: What is the recurrence?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 8T(n/2) + cn^2$$

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 8T(n/2) + cn^2 = O\left(n^{\lg 8}\right) = O\left(n^3\right)$$

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

TH16: What is the recurrence?

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

What is the recurrence?

$$T(n) \leq 7T(n/2) + cn^2 = O\left(n^{\lg 7}\right) = O\left(n^{2.8074}\right)$$

Divide & Conquer v2

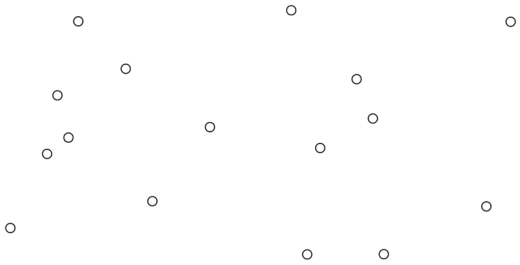
$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Current Champ: $O(n^{2.373})$

Virginia Vassilevska Williams, MIT

Closest Pairs*

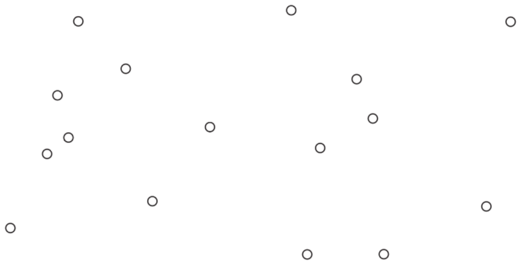
Finding the Closes Pair of Points



Problem

Given a set of n points, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, in the plane. Find the closest pair. That is, solve $\arg\min_{(p_i,p_j)\in\mathcal{P}}\{d(p_i, p_j)\}$, where $d(\cdot,\cdot)$ is the Euclidean distance.

Finding the Closes Pair of Points



Problem

Given a set of n points, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, in the plane. Find the closest pair. That is, solve $\arg\min_{(p_i,p_j)\in\mathcal{P}}\{d(p_i, p_j)\}$, where $d(\cdot,\cdot)$ is the Euclidean distance.

What is the $O(n^2)$ solution?

1-d Version

1-d Closest Pair

1-d Version

1-d Closest Pair

The points are on the line.

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points $(O(n \log n))$.

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points ($O(n \log n)$).
- Walk through sorted points and find minimum pair

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points ($O(n \log n)$).
- Walk through sorted points and find minimum pair ($O(n)$).

2-d Closest Pair

Divide and Conquer

1. Divide: Split point set (in half?).

2-d Closest Pair

Divide and Conquer

1. Divide: Split point set (in half?).
2. Conquer: Find closest pair in each partition.

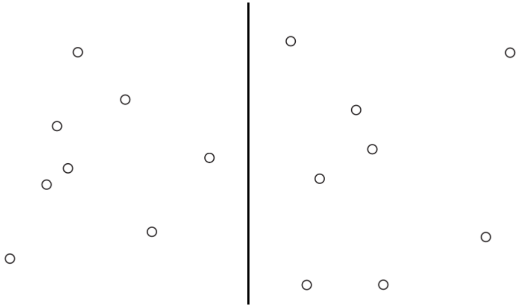
2-d Closest Pair

Divide and Conquer

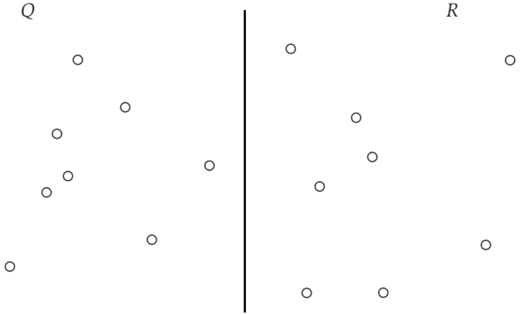
1. Divide: Split point set (in half?).
2. Conquer: Find closest pair in each partition.
3. Combine: Merge the solutions.

1. Divide: Split the Points

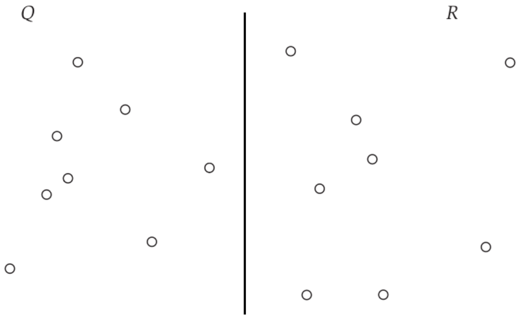
1. Divide: Split the Points



1. Divide: Split the Points



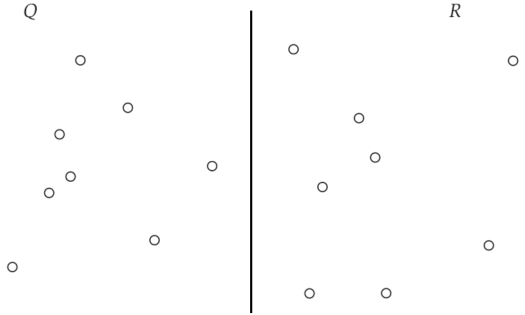
1. Divide: Split the Points



Definitions

- $\mathcal{P}_x$: Points sorted by x-coordinate.
- $\mathcal{P}_y$: Points sorted by y-coordinate.
- Q (resp. R) is left (resp. right) half of $\mathcal{P}_x$.

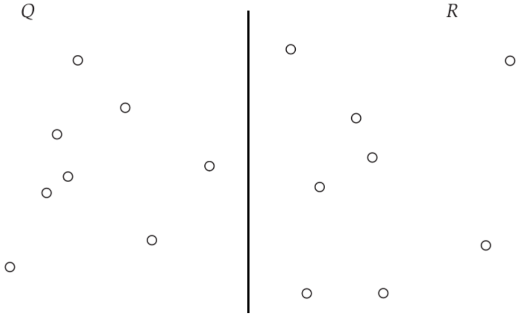
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_Y without resorting.

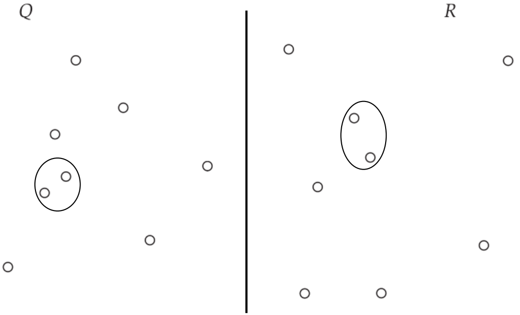
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_Y without resorting.
- Running time for this:

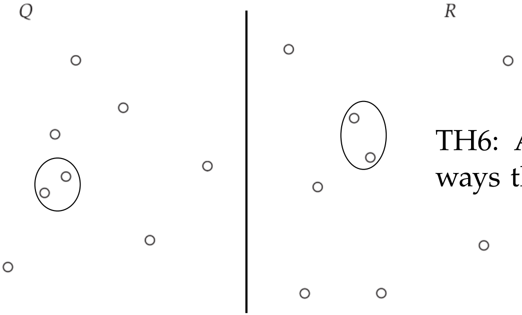
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_Y without resorting.
- Running time for this: $O(n)$.
- Let (q_0^*, q_1^*) and (r_0^*, r_1^*) be closest pairs in Q and R.

3. Combine the Solutions.

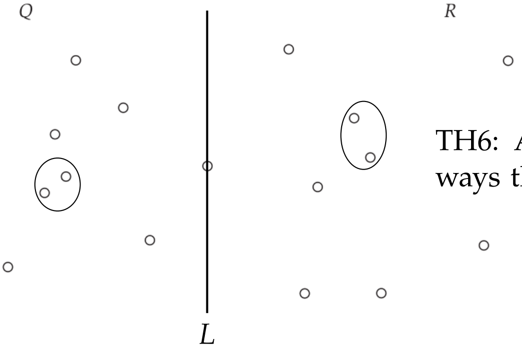


TH6: Are one of these always the minimum of $\mathcal{P}$?

Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_Y without resorting.
- Running time for this:
- Let (q_0^*, q_1^*) and (r_0^*, r_1^*) be closest pairs in Q and R.

3. Combine the Solutions.

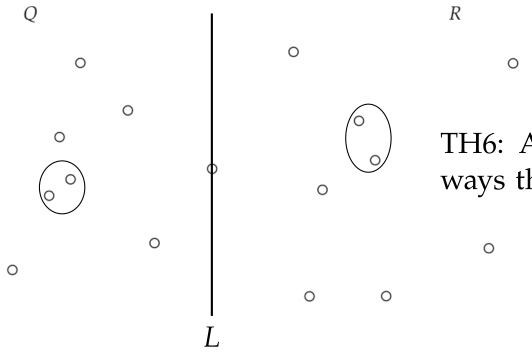


TH6: Are one of these always the minimum of $\mathcal{P}$?

Claim 1

[TopHat] Let $\delta := \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$*. If there exists a* $q \in Q$ *and an* $r \in R$ *for which* $d(q, r) < \delta$*, then each of* q *and* r *are within* $\square$ *of* L*.*

3. Combine the Solutions.

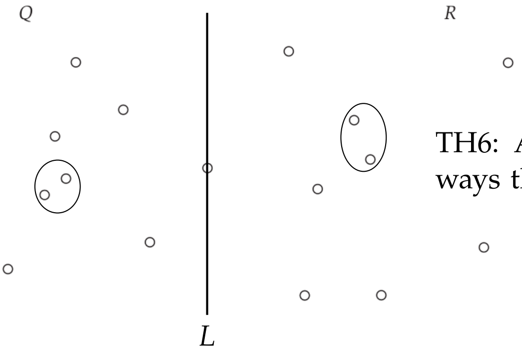


TH6: Are one of these always the minimum of $\mathcal{P}$?

Claim 1

Let $\delta := \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$. If there exists a $q \in Q$ and an $r \in R$ for which $d(q, r) < \delta$, then each of q and r are within δ of L.

3. Combine the Solutions.



TH6: Are one of these always the minimum of $\mathcal{P}$?

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.

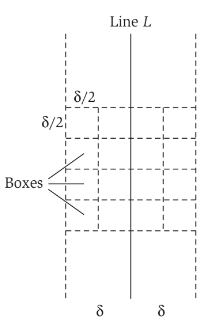
□

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



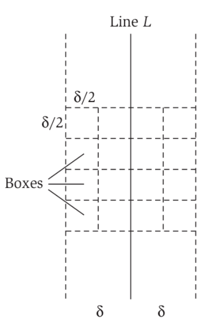
□

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.

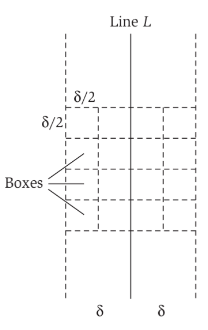
□

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s,s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.

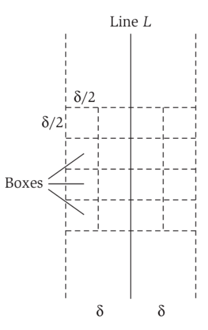
□

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.
- By way of contradiction, say $d(s, s') < \delta$ and s and s' separated by 16 positions.

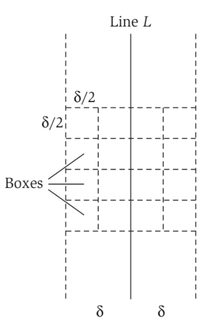
□

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.
- By way of contradiction, say $d(s, s') < \delta$ and s and s' separated by 16 positions.
- By counting argument, s and s' are separated by 3 rows which is at least $3\delta/2$. $\square$

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.
    - For each $p \in S$, check the distance to each of next 15 points in S_y.

3. Combine the Solutions.

Lemma 3

Let S be the set of points wihtin δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.
    - For each $p \in S$, check the distance to each of next 15 points in S_y.
- If $d(s, s') < \delta$, return (s, s')
- else return min of (q_0^*, q_1^*) and (r_0^*, r_1^*).

Completing the Analysis

Correctness of the Algorithm

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- TH: How many recursive calls?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- TH: What is the size of the recursive calls?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
    - $15 \cdot |S| = O(n)$.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
    - $15 \cdot |S| = O(n)$.
- TH: What is the recurrence?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
  - $15 \cdot |S| = O(n)$.
- What is the recurrence?

$$T(n) \leq 2T(n/2) + cn = O(n \log n) \ .$$

Max Subarray*

Max Subarray

Problem

Given an array A of integers, find the contiguous subarray of A of maximum sum.

Max Subarray

Problem

Given an array A of integers, find the contiguous subarray of A of maximum sum.

Exercise – Teams of 3 or so

- Solve the problem in $\Theta(n^2)$.
- Solve the problem in $O(n \log n)$.
- Prove correctness and complexity.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarrays

Input : Array A of n ints.
Output: Max subarray in A.

Let M be an empty array
for $i := 1$ *to* $len(A)$ **do**
  for $j := i$ *to* $len(A)$ **do**
    if $sum(A[i..j]) > sum(M)$ **then**
      $M := A[i..j]$
    end
  end
end
return M

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarra

Input : Array A of n ints.
Output: Max subarray in A.

Let M be an empty array
for $i := 1$ *to* $len(A)$ **do**
  for $j := i$ *to* $len(A)$ **do**
    if $sum(A[i..j]) > sum(M$
      $M := A[i..j]$
    end
  end
end
return M

Analysis

- Correct: Checks all possible contiguous subarrays.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarra

Input : Array A of n ints.
Output: Max subarray in A.

Let M be an empty array
for $i := 1$ *to* $len(A)$ **do**
  for $j := i$ *to* $len(A)$ **do**
    if $sum(A[i..j]) > sum(M$
      $M := A[i..j]$
    end
  end
end
return M

Analysis

- Correct: Checks all possible contiguous subarrays.
- Complexity:
  - Re-calculating the sum will make it $O(n^3)$. Key is to calculate the sum as you iterate.
  - For each i, check $n - i + 1$ ends. Overall:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2).$$

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.
$A_1 :=$ MaxSubarray(Front-half of A)
$A_2 :=$ MaxSubarray(Back-half of A)
$M :=$MidMaxSubarray(A)
return *Array with max sum of* $\{A_1, A_2, M\}$

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.

$A_1 :=$ MaxSubarray(Front-half of A)
$A_2 :=$ MaxSubarray(Back-half of A)
$M :=$MidMaxSubarray(A)
return *Array with max sum of* $\{A_1, A_2, M\}$

Algorithm: MidMaxSubarray

Input : Array A of n ints.
Output: Max subarray that crosses midpoint A.

$m :=$ mid-point of A
$L :=$ max subarray in $A[i, m-1]$ for $i = m-1 \to i$
$R :=$ max subarray in $A[m, j]$ for $j = m \to n$
return $L \cup R$ `// subarray formed by combining` L `and` R.

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.
A_1 := MaxSubarray(Front-half of A)
A_2 := MaxSubarray(Back-half of A)
M :=MidMaxSubarray(A)
return *Array with max sum of* $\{A_1, A_2, M\}$

Analysis

- Correctness: By induction, A_1 and A_2 are max for subarray and M is max mid-crossing array.
- Complexity: Same recurrence as MergeSort.

Appendix

References

Image Sources I

https://brand.wisc.edu/web/logos/

https://people.csail.mit.edu/virgi/