Lecture 10: Divide & Conquer; String Matching I

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: November 5, 2024

Announcements

1. **NO QUIZ THIS WEEK!**
2. Programming Assignment Posted
   - Due Wednesday, 13 November
3. Attendance Code:

780493

Meeting Goals

- Discuss more Divide & Conquer algorithms
  - Order Statistics
  - Majority
  - Closest Pair of Points
- Introduce the **String Matching** problem
  - Problem definition
  - Elementary algorithm

Divide & Conquer

Previously: Divide & Conquer Strategy

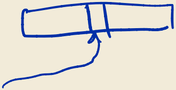
Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Examples (so far):

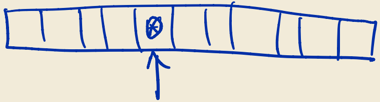
- MergeSort: divide an array *by index* to sort
  - $O(n \log n)$ time
- QuickSort: divide an array *by value* to sort
  - $O(n \log n)$ time
- BinarySearch: divide a *sorted* array to search it
  - $O(\log n)$ time



Three More Problems

Problem 1. *k*-Selection:

- Given an array a of n numbers, find the kth largest number



v

Such that on sorting a

v would be in index k

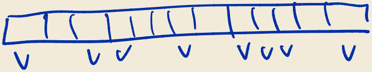
Three More Problems

Problem 1. k-Selection:

- Given an array a of n numbers, find the kth largest number

Problem 2. Majority:

- Given an array a of n items, is there an item that is repeated more than $> n/2$ times?



Three More Problems

Problem 1. k-Selection:

- Given an array a of n numbers, find the kth largest number

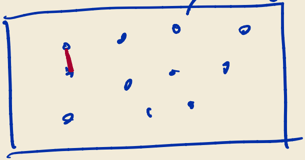
Problem 2. Majority:

- Given an array a of n items, is there an item that is repeated more than $> n/2$ times?

Problem 3. Closest Points in the Plane

- Given n points $p_1, p_2, \ldots, p_n$ in the plane, which *pair* of points p_i, p_j are closest to one another?

$p_i = (x_i, y_i)$



k-Selection

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

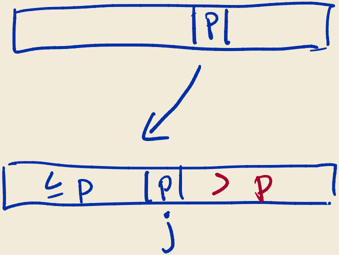
Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

Modify QuickSort!

- Choose pivot p
- Perform split
- *only recurse on half that contains kth smallest value*
    - this will be the half that contains index k
- Random pivot selection
  $\Longrightarrow$ $O(n)$ expected time!



if $j \geq k \Rightarrow k^{th}$ smallest is left half

if $j < k \Rightarrow k^{th}$ smallest val on right side

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

Modify QuickSort!

- Choose pivot p
- Perform split
- *only recurse on half that contains kth smallest value*
  - this will be the half that contains index k
- Random pivot selection $\implies O(n)$ expected time!

```
procedure QuickSelect(a, min, max, k)
    if max − min ≤ 1 then
        return a[min]
    end if
    p ← SelectPivot(a, min, max)
    j ← Split(a, min, max, p)
    if j = k then
        return a[k]
    else if j < k then
        QuickSelect(a, j + 1, max, k)
    else
        QuickSelect(a, min, j − 1, k)
    end if
end procedure
```

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

PollEverywhere Question

What is the *worst case* running time of QUICKSELECT on an array of n elements?

pollev.com/comp526

```
procedure QUICKSELECT(a, min, max, k)
    if max − min ≤ 1 then
        return a[min]
    end if
    p ← SELECTPIVOT(a, min, max)
    j ← SPLIT(a, min, max, p)
    if j = k then
        return a[k]
    else if j < k then
        QUICKSELECT(a, j + 1, max, k)
    else
        QUICKSELECT(a, min, j − 1, k)
    end if
end procedure
```

$k=1$

P_1 = first pivot = largest val

$|\leftarrow n-1 \rightarrow|$ P_1 → recurse on first $n-1$ elts. ... steps. $\Rightarrow \Omega(n^2)$

Deterministic k-Selection?

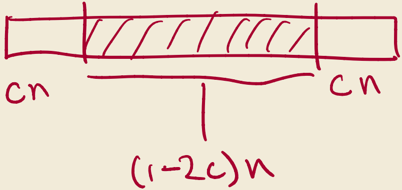
Question. Can we perform k-selection with a **worst case** $O(n)$ running time?

Deterministic k-Selection?

Question. Can we perform k-selection with a **worst case** $O(n)$ running time?

Idea. What if we can select better pivots?

- Suppose we can *guarantee* that our pivot is "good enough:"
  - rank of p is between cn and $(1-c)n$ for $c > 0$
- How many recursive calls until we're done?



Can we guarantee a pivot always is here?

Deterministic k-Selection?

Question. Can we perform k-selection with a **worst case** $O(n)$ running time?

Idea. What if we can select better pivots?

- Suppose we can *guarantee* that our pivot is "good enough:"
  - rank of p is between cn and $(1-c)n$ for $c > 0$
- How many recursive calls until we're done?
  - each recursive call has size at most $(1-2c)n$
  - ℓ recursive calls $\implies$ size at most $(1-2c)^{\ell} n$
  - $\implies$ done after $\ell = O(\log n)$ levels of recursion
- What is overall running time?

Deterministic k-Selection?

Question. Can we perform k-selection with a **worst case** $O(n)$ running time?

Idea. What if we can select better pivots?

- Suppose we can *guarantee* that our pivot is "good enough:"
  - rank of p is between cn and $(1-c)n$ for $c > 0$
- How many recursive calls until we're done?
  - each recursive call has size at most $(1-2c)n$
  - ℓ recursive calls $\implies$ size at most $(1-2c)^{\ell} n$
  - $\implies$ done after $\ell = O(\log n)$ levels of recursion
- What is overall running time?
  - $\boxed{Cn} + \boxed{(1-2c)}Cn + \boxed{(1-2c)^2}Cn + \cdots = \boxed{O(n)}$

$$= Cn\left(1 + \underbrace{(1-2c)}_{a} + \underbrace{(1-2c)^2}_{a^2} + \cdots\right)$$

$$= Cn \boxed{\sum_{i=0}^{?} a^i} < \frac{1}{1-a} = \text{const} \Rightarrow O(n).$$

Deterministic k-Selection?

Question. Can we perform k-selection with a **worst case** $O(n)$ running time?

Idea. What if we can select better pivots?

- Suppose we can *guarantee* that our pivot is "good enough:"
  - rank of p is between cn and $(1-c)n$ for $c > 0$
- How many recursive calls until we're done?
  - each recursive call has size at most $(1-2c)n$
  - ℓ recursive calls $\implies$ size at most $(1-2c)^\ell n$
  - $\implies$ done after $\ell = O(\log n)$ levels of recursion
- What is overall running time?
  - $Cn + (1-2c)Cn + (1-2c)^2 Cn + \cdots = O(n)$

But how can we find a good pivot *deterministically*?

- Need to find pivots close to the median…
- Median is (special case) of k selection!

Median of Medians Strategy

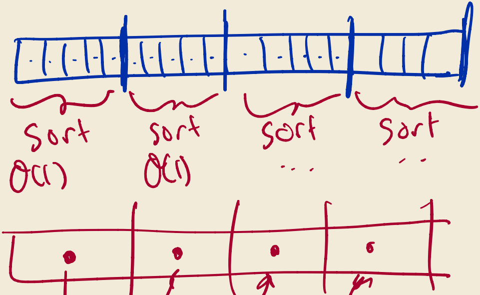
Strategy. To find a good pivot:

- Find a *smaller* set of values whose *median* is a good pivot
- Recursively find the median of the smaller set of values

Median of Medians Strategy

Strategy. To find a good pivot:

- Find a *smaller* set of values whose *median* is a good pivot
- Recursively find the median of the smaller set of values
- Consider blocks of size 5
  - sort each block
  - find the block median
- Claim: median of medians is a good pivot:

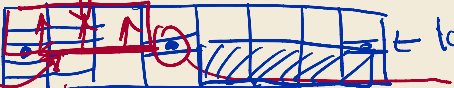


smaller median

$\Rightarrow \frac{n}{5}$ of these

all values here are smaller than x_m

What can we say about median of medians?



$\frac{n}{10}$ medians here

$\Rightarrow \frac{3n}{10}$ smaller values

← largest x_m

med of medians

Median of Medians Strategy

Strategy. To find a good pivot:

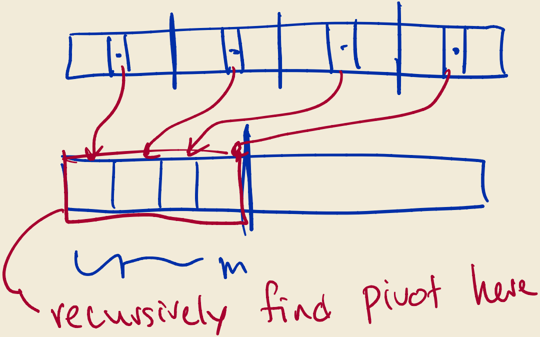
- Find a *smaller* set of values whose *median* is a good pivot
- Recursively find the median of the smaller set of values
- Consider blocks of size 5
  - sort each block
  - find the block median
- Claim: median of medians is a good pivot:
  - at least $\frac{3}{10}$-fraction is excluded

Median of Medians Strategy

Strategy. To find a good pivot:

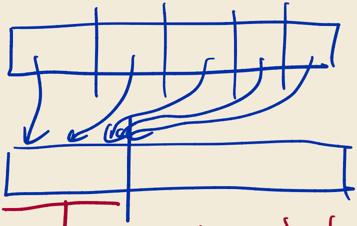
- Find a *smaller* set of values whose *median* is a good pivot
- Recursively find the median of the smaller set of values
- Consider blocks of size 5
  - sort each block
  - find the block median
- Claim: median of medians is a good pivot:
  - at least $\frac{3}{10}$-fraction is excluded

```
procedure SelectPivot(a, ℓ, r)
    m ← n/5
    for i = 0, 1, ..., m − 1 do
        Sort(a[5i ... 5i + 4])
        Swap(a, i, 5i + 2)
    end for
    return QuickSelect(a, 0, m, (m − 1)/2)
end procedure
```

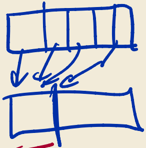


Median of Medians Strategy

Illustration:



quick select here to find median



quick select to find median..

procedure SELECTPIVOT(a, ℓ, r)
    $m \leftarrow n/5$
    for $i = 0, 1, \ldots, m-1$ **do**
        SORT$(a[5i \ldots 5i+4])$
        SWAP$(a, i, 5i+2)$
    end for
    return QUICKSELECT$(a, 0, m, (m-1)/2)$
end procedure
procedure QUICKSELECT(a, ℓ, r, k)
    if $r - \ell \leq 1$ **return** $a[l]$
    $b \leftarrow$ SELECTPIVOT(a, ℓ, r)
    $j \leftarrow$ SPLIT$(a, \ell, r, a[b])$
    if $j = k$ **then**
        return $a[j]$
    else if $j < k$ **then**
        QUICKSELECT$a, j+1, r, k-j-1$
    else
        QUICKSELECT$(a, 0, j, k)$
    end if
end procedure

Median of Medians Strategy

time to find pivot

Analysis.

Running time $T(n)$ satisfies

split

$$T(n) \le \boxed{Cn} + \boxed{T\left(\frac{1}{5}n\right)} + \boxed{T\left(\frac{7}{10}n\right)}$$

$$\le Cn + T\left(\frac{1}{5}n + \frac{7}{10}n\right)$$

$$\le Cn + T\left(\frac{9}{10}n\right)$$

Therefore, $T(n) = O(n)$.

Careful! Only true because final sol'n is $\Theta(n)$.

remaining values after finding a good pivot

```
procedure SelectPivot(a, ℓ, r)
    m ← n/5
    for i = 0, 1, ..., m − 1 do
        Sort(a[5i...5i + 4])
        Swap(a, i, 5i + 2)
    end for
    return QuickSelect(a, 0, m, (m − 1)/2)
end procedure
procedure QuickSelect(a, ℓ, r, k)
    if r − ℓ ≤ 1 return a[l]
    b ← SelectPivot(a, ℓ, r)
    j ← Split(a, ℓ, r, a[b])
    if j = k then
        return a[j]
    else if j < k then
        QuickSelecta, j + 1, r, k − j − 1
    else
        QuickSelect(a, 0, j, k)
    end if
end procedure
```

Median of Medians Strategy

Conclusion. The Median of Medians strategy allows us to

- solve k-selection in $O(n)$ time, worst case
- sort in $O(n \log n)$ time, worst case too
  - use k selection as a sub-routine for SELECTPIVOT in QUICKSORT

Note. Randomized variants tend to be more efficient in practice.

```
procedure SELECTPIVOT(a, ℓ, r)
    m ← n/5
    for i = 0, 1, ..., m − 1 do
        SORT(a[5i ... 5i + 4])
        SWAP(a, i, 5i + 2)
    end for
    return QUICKSELECT(a, 0, m, (m − 1)/2)
end procedure
procedure QUICKSELECT(a, ℓ, r, k)
    if r − ℓ ≤ 1 return a[l]
    b ← SELECTPIVOT(a, ℓ, r)
    j ← SPLIT(a, ℓ, r, a[b])
    if j = k then
        return a[j]
    else if j < k then
        QUICKSELECTa, j + 1, r, k − j − 1
    else
        QUICKSELECT(a, 0, j, k)
    end if
end procedure
```

Majority

Majority

Problem 2. Majority:

- Given an array a of n items, is there an item that is repeated more than $n/2$ times?

Naive Solution

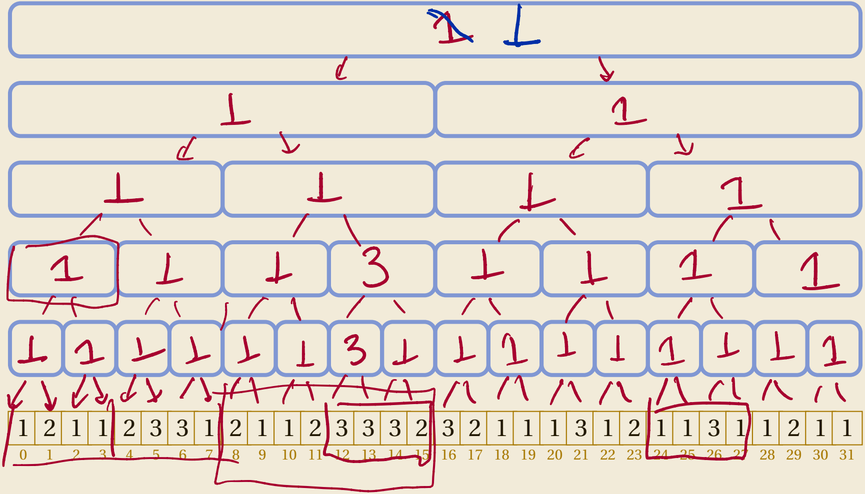
- Iterate over elements and compare each element to all others to see if occurs at least $n/2$ times
- Takes $\Theta(n^2)$ time

Observation. If a value m is a majority, then m must either be a majority in $a[0 \ldots n/2]$ or $a[n/2+1 \ldots n-1]$ as well.

- Split a in half
- Recursively find candidate majority m_ℓ and m_r for halves
- Check to see if either is a majority

Divide & Conquer Majority Illustration

Todo: fix me!



Divide & Conquer Majority in Code

```
procedure IsMajority(a, ℓ, r, v)
    count ← 0
    for i = ℓ, ℓ + 1, ..., r do
        if a[i] = v then
            count ← count + 1
        end if
    end for
    return count > (r − ℓ + 1)/2
end procedure
procedure Majority(a, ℓ, r)
    if ℓ − r < 1 return a[ℓ]
    j ← (r − ℓ)/2
    v_ℓ ← Majority(a, ℓ, j)
    v_r ← Majority(a, j + 1, r)
    if IsMajority(a, ℓ, r, v_ℓ) then
        return v_ℓ
    else if IsMajority(a, ℓ, r, v_r) then
        return v_r
    end if
    return ⊥
end procedure
```

is v the majority in a between ind ℓ and r

split in half and check halves for majority

Divide & Conquer Majority in Code

PollEverywhere Question

What is the *worst case* running time of MAJORITY on an array of n elements?

pollev.com/comp526

```
procedure IsMajority(a, ℓ, r, v)
    count ← 0
    for i = ℓ, ℓ+1, ..., r do
        if a[i] = v then
            count ← count + 1
        end if
    end for
    return count > (r − ℓ + 1)/2
end procedure
procedure Majority(a, ℓ, r)
    if ℓ − r < 1 return a[ℓ]
    j ← (r − ℓ)/2
    v_ℓ ← Majority(a, ℓ, j)
    v_r ← Majority(a, j+1, r)
    if IsMajority(a, ℓ, r, v_ℓ) then
        return v_ℓ
    else if IsMajority(a, ℓ, r, v_r) then
        return v_r
    end if
    return ⊥
end procedure
```

Divide & Conquer Majority in Code

$\Theta(r-\ell)$

Analysis.

- Almost identical to MergeSort
- Each call to IsMajority(a, ℓ, r, v) takes time $\Theta(\ell - r)$
- Running time $T(n)$ satisfies $\boxed{T(n) \le 2T(n/2) + \Theta(n)}$
- Solve recursion $\implies$ done!

$\Rightarrow \Theta(n \log n).$

```
procedure IsMajority(a, ℓ, r, v)
    count ← 0
    for i = ℓ, ℓ+1, ..., r do
        if a[i] = v then
            count ← count + 1
        end if
    end for
    return count > (r − ℓ + 1)/2
end procedure
procedure Majority(a, ℓ, r)
    if ℓ − r < 1 return a[ℓ]
    j ← (r − ℓ)/2
    v_ℓ ← Majority(a, ℓ, j)                 T(n/2)
    v_r ← Majority(a, j+1, r)               T(n/2)
    if IsMajority(a, ℓ, r, v_ℓ) then        Θ(n)
        return v_ℓ
    else if IsMajority(a, ℓ, r, v_r) then
        return v_r
    end if
    return ⊥
end procedure
```

Divide & Conquer Majority in Code

Analysis.

- Almost identical to MERGESORT
- Each call to ISMAJORITY(a, ℓ, r, v) takes time $\Theta(\ell - r)$
- Running time $T(n)$ satisfies $T(n) \leq 2T(n/2) + \Theta(n)$
- Solve recursion $\implies$ done!

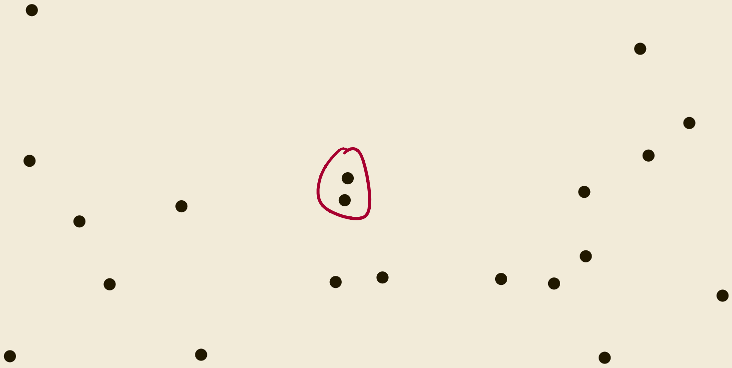
Challenge. Devise an algorithm that finds the majority in $\Theta(n)$ time (worst case). (Hint: don't use Divide & Conquer)

```
procedure IsMajority(a, ℓ, r, v)
    count ← 0
    for i = ℓ, ℓ+1, ..., r do
        if a[i] = v then
            count ← count + 1
        end if
    end for
    return count > (r − ℓ + 1)/2
end procedure
procedure Majority(a, ℓ, r)
    if ℓ − r < 1 return a[ℓ]
    j ← (r − ℓ)/2
    v_ℓ ← Majority(a, ℓ, j)
    v_r ← Majority(a, j+1, r)
    if IsMajority(a, ℓ, r, v_ℓ) then
        return v_ℓ
    else if IsMajority(a, ℓ, r, v_r) then
        return v_r
    end if
    return ⊥
end procedure
```

Closest Points in the Plane

Closest Points in the Plane

Problem 3. Given n points $p_1, p_2, \ldots, p_n$ in the plane, which *pair* of points p_i, p_j are closest to one another?



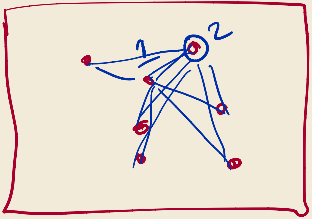
Closest Points in the Plane

Problem 3. Given n points $p_1, p_2, \ldots, p_n$ in the plane, which *pair* of points p_i, p_j are closest to one another?

Naive Strategy suggested by GenAI:

- Compute distances between all pairs of points

procedure NaiveMinDist(p)
    $d \leftarrow \infty$
    for $i = 1, 2, \ldots, n-1$ **do**
        for $j = 0, 1, \ldots, i-1$ **do**
            if Dist($p[i], p[j]$) $< d$ **then**
                $d \leftarrow$ Dist($p[i], p[j]$)
            end if
        end for
    end for
    return d
end procedure



Closest Points in the Plane

Problem 3. Given n points $p_1, p_2, \ldots, p_n$ in the plane, which *pair* of points p_i, p_j are closest to one another?

Assume: $\text{Dist}(p, q) = \Theta(1)$.

PollEverywhere Question

What is the worst case running time of NaiveMinDist on a set of n points in the plane?

pollev.com/comp526

```
procedure NaiveMinDist(p)
    d ← ∞
    for i = 1, 2, ..., n − 1 do
        for j = 0, 1, ..., i − 1 do
            if Dist(p[i], p[j]) < d then
                d ← Dist(p[i], p[j])
            end if
        end for
    end for
    return d
end procedure
```

$\Theta(1) = c$

i times $\Rightarrow$ time $c \cdot i$

n iterations

Overall: $c + 2c + 3c + 4c \cdots + (n-1)c$

$= \Theta(n^2)$.

Closest Points in the Plane

Problem 3. Given n points $p_1, p_2, \ldots, p_n$ in the plane, which *pair* of points p_i, p_j are closest to one another?

Naive Strategy suggested by GenAI:

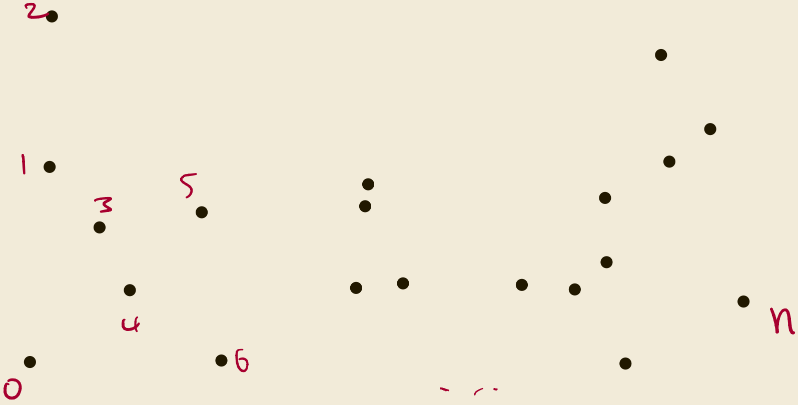
- Compute distances between all pairs of points

```
procedure NaiveMinDist(p)
    d ← ∞
    for i = 1, 2, ..., n − 1 do
        for j = 0, 1, ..., i − 1 do
            if Dist(p[i], p[j]) < d then
                d ← Dist(p[i], p[j])
            end if
        end for
    end for
    return d
end procedure
```

Question. How could we use **Divide & Conquer** to improve on this running time?

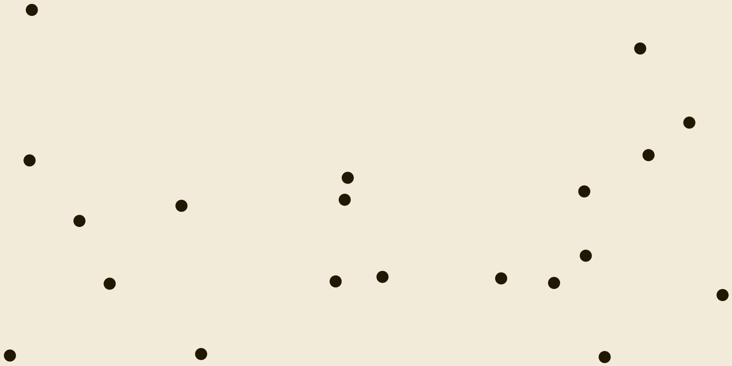
Closest Points: Divide & Conquer

Step 1. split the array according to *x*-coordinate



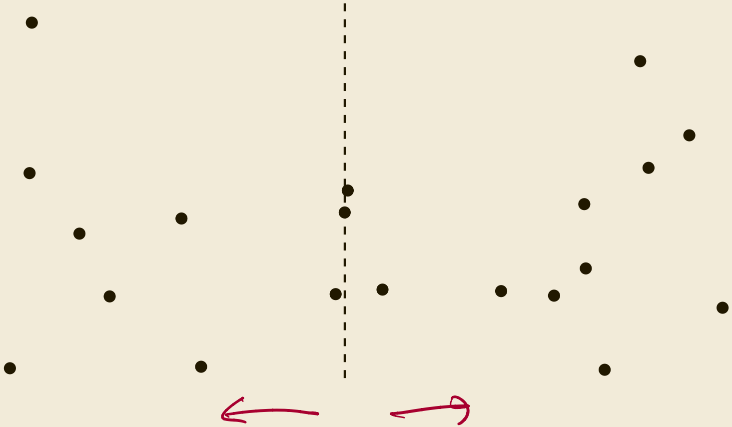
Closest Points: Divide & Conquer

Step 1a. sort the array by x coordinate



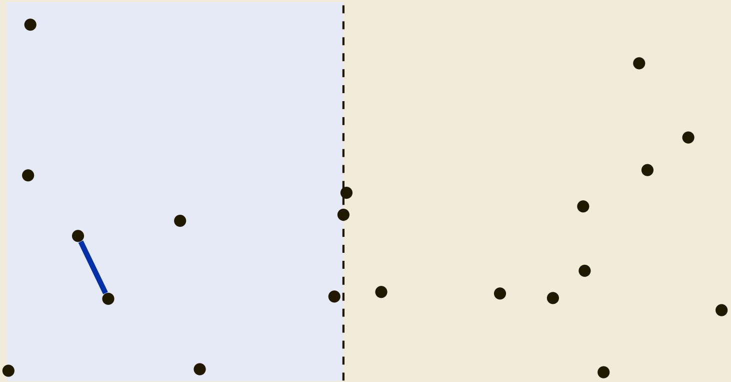
Closest Points: Divide & Conquer

Step 1b. find median according to x coordinate, p_m



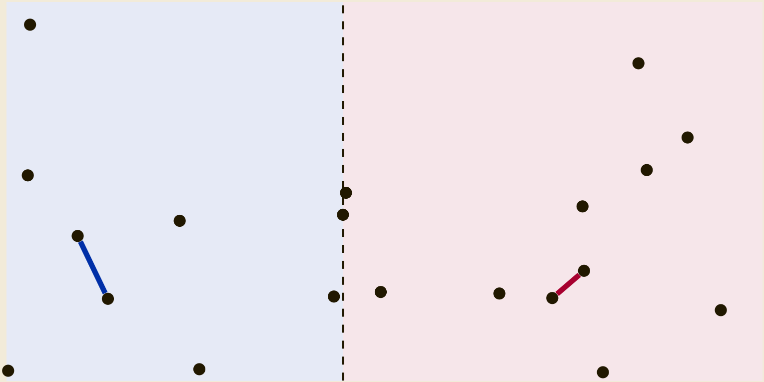
Closest Points: Divide & Conquer

Step 2a. (recursively) solve the problem for left half



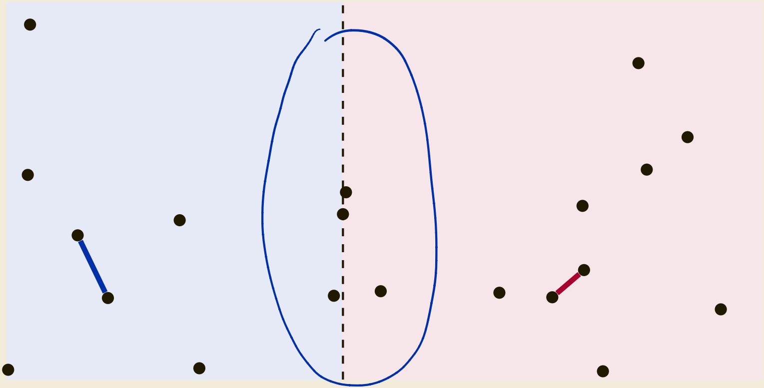
Closest Points: Divide & Conquer

Step 2b. (recursively) solve the problem for right half



Closest Points: Divide & Conquer

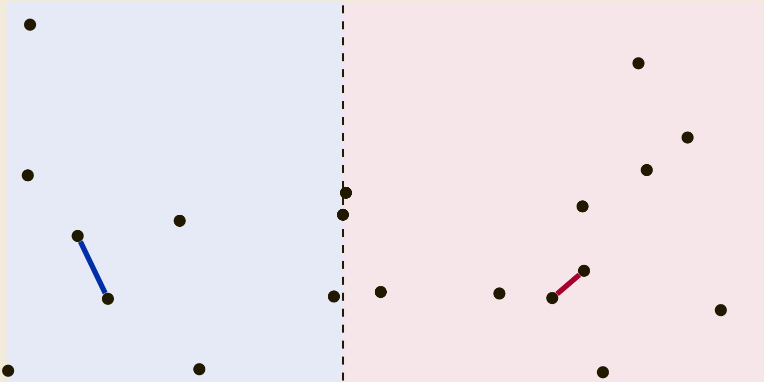
Step 3. merge solutions together



Deal w/ points
that cross boundary

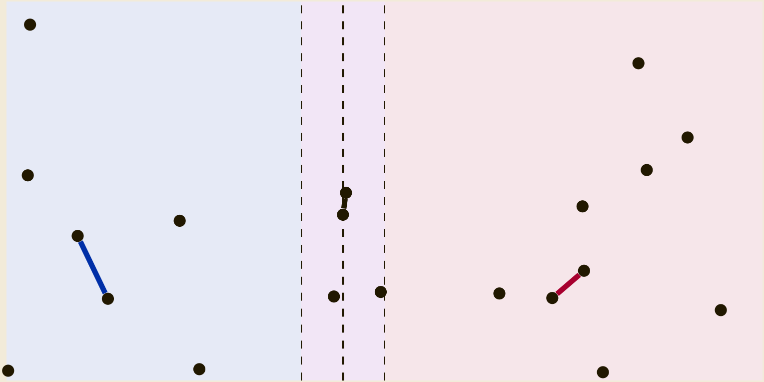
Closest Points: Divide & Conquer

Step 3. merge solutions together . . . but **how?**



Closest Points: Divide & Conquer

Critical Analysis. What happens in the middle strip?

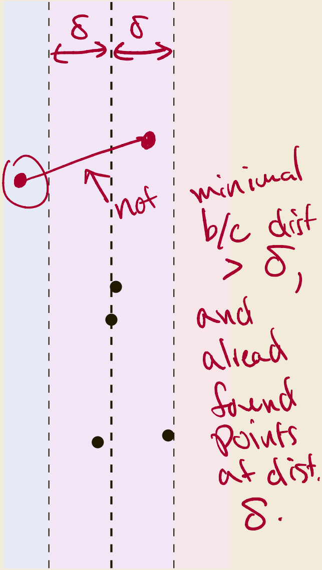


How wide to guarantee that min dist must be within strip (or on one side)?

Analysis of the Middle Strip

Suppose:

- d_ℓ is minimal distance on the left
- d_r is minimal distance on the right
- $\delta = \min\{d_\ell, d_r\}$
- x_m is the median x-coordinate among points

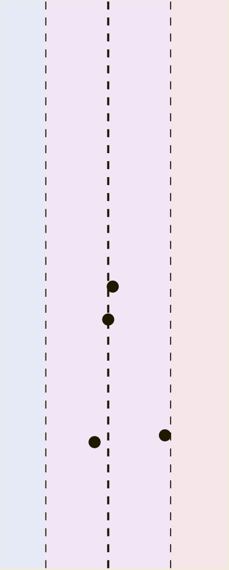


Analysis of the Middle Strip

Suppose:

- d_ℓ is minimal distance on the left
- d_r is minimal distance on the right
- $\delta = \min\{d_\ell, d_r\}$
- x_m is the median x-coordinate among points

Claim 1. If p is in left half and q is on right have with $\textsc{Dist}(p_i, p_j) < \delta$, then $x_m - \delta < x_i \leq x_m$ and $x_m \leq x_j \leq x_m + \delta$.



Analysis of the Middle Strip

Suppose:

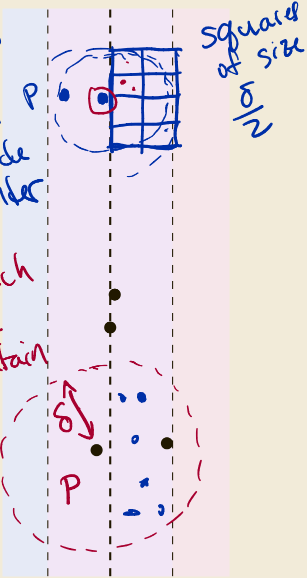
- d_ℓ is minimal distance on the left
- d_r is minimal distance on the right
- $\delta = \min\{d_\ell, d_r\}$
- x_m is the median x-coordinate among points

Claim 1. If p is in left half and q is on right have with $\textsc{Dist}(p_i, p_j) < \delta$, then $x_m - \delta < x_i \le x_m$ and $x_m \le x_j \le x_m + \delta$.

Claim 2. With p as above, there are at most 8 points q on the right side with $\textsc{Dist}(p, q) \le \delta$.

at most 8 squares intersect disc on p for side of center line

but each square can contain at most 1 point.



Analysis of the Middle Strip

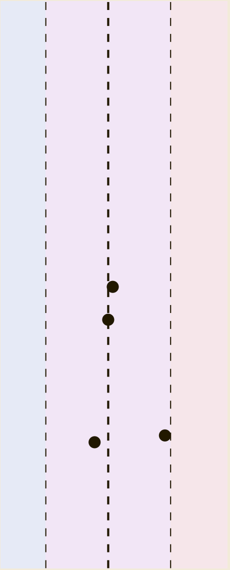
Suppose:

- d_ℓ is minimal distance on the left
- d_r is minimal distance on the right
- $\delta = \min\{d_\ell, d_r\}$
- x_m is the median x-coordinate among points

Claim 1. If p is in left half and q is on right have with $\textsc{Dist}(p_i, p_j) < \delta$, then $x_m - \delta < x_i \le x_m$ and $x_m \le x_j \le x_m + \delta$.

Claim 2. With p as above, there are at most 8 points q on the right side with $\textsc{Dist}(p, q) \le \delta$.

Consequence. We only need to make $O(n)$ further distance computations to compute overall minimum distance.



Putting it Together

Algorithm Sketch. Find the closest pair of points among $p_1, p_2, \ldots, p_n$ in the plane:

1. Sort points by x-coordinate, x_m is the median value. ← $O(n \log n)$
2. Recursively sort left and right halves. $2T(n/2)$
3. Set δ to be the minimum distance on either half. – $O(1)$
4. Consider points within distance δ of median line, and compute distances across the halves. $O(n)$
   - this can be done in $O(n)$ time
5. Report the smallest distance found.

Putting it Together

Algorithm Sketch. Find the closest pair of points among $p_1, p_2, \ldots, p_n$ in the plane:

1. Sort points by x-coordinate, x_m is the median value.
2. Recursively sort left and right halves.
3. Set δ to be the minimum distance on either half.
4. Consider points within distance δ of median line, and compute distances across the halves.
    - this can be done in $O(n)$ time
5. Report the smallest distance found.

Running time analysis.

- Preprocessing takes $O(n \log n)$ to sort the points.
- The main algorithm running time satisfies the recursion $T(n) \leq 2T(n/2) + O(n)$

$\Longrightarrow$ overall running time is $O(n \log n)$.

Concluding Thoughts

Divide & Conquer is a powerful algorithm design strategy.
Efficiency improvement over naive solutions:

- Sorting $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- k-Selection $\Theta(n^2) \longrightarrow \Theta(n)$
- Majority $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- Closest points in the plane $\Theta(n^2) \longrightarrow \Theta(n \log n)$

Concluding Thoughts

Divide & Conquer is a powerful algorithm design strategy.
Efficiency improvement over naive solutions:

- Sorting $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- k-Selection $\Theta(n^2) \longrightarrow \Theta(n)$
- Majority $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- Closest points in the plane $\Theta(n^2) \longrightarrow \Theta(n \log n)$

Other applications:

- Matrix multiplication (Strassen's algorithm):
  $\Theta(n^3) \longrightarrow \Theta(n^{\log_2 7 + o(1)}) \approx \Theta(n^{2.807})$
- Integer multiplication: $\Theta(B^2) \longrightarrow \Theta(B^{\log_2 3}) \longrightarrow \Theta(B \log B)$
- Fast Fourier Transform: $\Theta(n^2) \longrightarrow \Theta(n \log n)$

Concluding Thoughts

Divide & Conquer is a powerful algorithm design strategy.
Efficiency improvement over naive solutions:

- Sorting $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- k-Selection $\Theta(n^2) \longrightarrow \Theta(n)$
- Majority $\Theta(n^2) \longrightarrow \Theta(n \log n)$
- Closest points in the plane $\Theta(n^2) \longrightarrow \Theta(n \log n)$

Other applications:

- Matrix multiplication (Strassen's algorithm): $\Theta(n^3) \longrightarrow \Theta(n^{\log_2 7 + o(1)}) \approx \Theta(n^{2.807})$
- Integer multiplication: $\Theta(B^2) \longrightarrow \Theta(B^{\log_2 3}) \longrightarrow \Theta(B \log B)$
- Fast Fourier Transform: $\Theta(n^2) \longrightarrow \Theta(n \log n)$

Other considerations:

- Practical because of **parallelism**

String Matching

String Matching: Motivation

Fundamental Problems. Given a (large) **text** $\boxed{T}$ and (small) **pattern** $\boxed{P}$:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

String Matching: Motivation

Fundamental Problems. Given a (large) **text** T and (small) **pattern** P:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

Example applications.

- Search on your computer: Ctrl + F

String Matching: Motivation

Fundamental Problems. Given a (large) **text** T and (small) **pattern** P:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

Example applications.

- Search on your computer: Ctrl + F
- Bioinformatics:
  - does a DNA sequence (T) contain a particular gene (P)?

String Matching: Motivation

Fundamental Problems. Given a (large) **text** T and (small) **pattern** P:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

Example applications.

- Search on your computer: Ctrl + F
- Bioinformatics:
  - does a DNA sequence (T) contain a particular gene (P)?
- Computer virus detection
  - does your hard drive store a known program?

String Matching: Motivation

Fundamental Problems. Given a (large) **text** T and (small) **pattern** P:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

Example applications.

- Search on your computer: Ctrl + F
- Bioinformatics:
  - does a DNA sequence (T) contain a particular gene (P)?
- Computer virus detection
  - does your hard drive store a known program?
- (Counter) Espionage
  - does a data transmission contain the phrase "`ATTACK AT DAWN?`"

String Matching: Motivation

Fundamental Problems. Given a (large) **text** T and (small) **pattern** P:

- Determine if T contains the pattern P.
- Find the *first occurrence* of P in T (if any)
- Fund the number of occurrences of P in T

Example applications.

- Search on your computer: Ctrl + F
- Bioinformatics:
  - does a DNA sequence (T) contain a particular gene (P)?
- Computer virus detection
  - does your hard drive store a known program?
- (Counter) Espionage
  - does a data transmission contain the phrase "`ATTACK AT DAWN?`"

Interesting parameters. $|T|$ is large ($\sim$ 1B), $|P|$ is relatively small ($\sim$ 1K)

Making Things Precise

Notation

- Σ is a finite **alphabet** or set of **characters**, $\sigma = |\Sigma|$
    - $\Sigma = \{0, 1\}$ is binary alphabet
    - $\Sigma = \{A, B, \ldots\}$ is Roman alphabet
    - $\Sigma = \cdots$ e.g., ASCII, Unicode,
- $\Sigma^n = \Sigma \times \Sigma \times \cdots \times \Sigma = \{(c_1, c_2, \ldots, c_n) \mid \text{each } c_i \in \Sigma\}$ = strings of exactly n characters
- $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ = all *finite* strings
- $\Sigma^+ = \bigcup_{n=0}^{\infty} \Sigma^n$ = all *nonempty* (finite) strings
- $\boxed{\varepsilon} \in \Sigma^0$ is the **empty string**
- for $S \in \Sigma^n$, $S[i]$ is ith character of S
- for $S, T \in \Sigma^*$, $\boxed{ST}$ is the **concatenation** of S and T
- for $S \in \Sigma^n$, $S[i..j] = S[i]\,S[i+1] \cdots S[j]$ is a **substring**
    - $S[0..j]$ is a **prefix**, $S[j..n-1]$ is a **suffix**
    - $S[i..j) = S[i..j-1] \implies S = S[0..n)$

The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m (typically $m \ll n$)

Output:

- The index of the **first occurrence** of P in T, or -1 if T does not contain P as a substring:
  - $\min\{i \mid T[i, i+m) = P\}$

Example.

- $T = 101100\boxed{1101}1101$
- $P_1 = 1101$

The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m (typically $m \ll n$)

Output:

- The index of the **first occurrence** of P in T, or -1 if T does not contain P as a substring:
  - $\min\{i \mid T[i, i+m) = P\}$

Example.

- $T = 1011001101 1101$
- $P_1 = 1101$
  - Output: $i \leftarrow 6$

The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m (typically $m \ll n$)

Output:

- The index of the **first occurrence** of P in T, or -1 if T does not contain P as a substring:
  - $\min\{i \mid T[i, i+m) = P\}$

Example.

- $T = 10110011011101$
- $P_1 = 1101$
  - Output: $i \leftarrow 6$
- $P_2 = 000$

The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m (typically $m \ll n$)

Output:

- The index of the **first occurrence** of P in T, or -1 if T does not contain P as a substring:
  - $\min\{i \mid T[i, i+m) = P\}$

Example.

- $T = 10110011011101$
- $P_1 = 1101$
  - Output: $i \leftarrow 6$
- $P_2 = 000$
  - Output: $i \leftarrow -1$

Brute Force Matching

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i+m) = P$?
- verify each character individually

procedure VerifyMatch(T, P, i)
    $j \leftarrow 0$
    while $j < m$ **do**
        if $T[i+j] \neq P[j]$ **then**
            return False
        end if
        $j \leftarrow j + 1$
    end while
    return True
end procedure

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i+m) = P$?
- verify each character individually

procedure VerifyMatch(T, P, i)
  $j \leftarrow 0$
  while $j < m$ **do**
    if $T[i+j] \neq P[j]$ **then**
      return False
    end if
    $j \leftarrow j + 1$
  end while
  return True
end procedure

Cost = number of comparisons made

PollEverywhere Question

What are the worst case and best case running times of VerifyMatch?

pollev.com/comp526

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i+m) = P$?
- verify each character individually

procedure VERIFYMATCH(T, P, i)
    $j \leftarrow 0$
    while $j < m$ **do**
        if $T[i+j] \neq P[j]$ **then**
            return FALSE
        end if
        $j \leftarrow j + 1$
    end while
    return TRUE
end procedure

Best and Worst Cases:

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

Cost = number of comparisons made

Brute force. Guess and check every value $i = 0, 1, \ldots, n - m - 1$

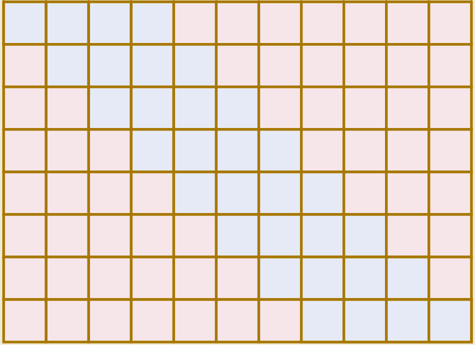
- Worst case running time is $\Theta(nm)$
  - What is example has cost $\Omega(nm)$?
- Best case cost is $\Theta(m)$

Brute Force Example

Example

- $T = abbbababbab$
- $P = abba$

0	1	2	3	4	5	6	7	8	9	10
a	b	b	b	a	b	a	b	b	a	b



```
procedure
BRUTEFORCEMATCH(T, P)
    for i = 0, 1, ..., n − m − 1 do
        if VERIFYMATCH(T, P, i) then
            return i
        end if
    end for
    return −1
end procedure
```

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)\ldots$

$\ldots$ but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters.*

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)\ldots$

$\ldots$ but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)\ldots$

$\ldots$but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*
- Which of these comparisons were unnecessary?
  - How can you search with fewer comparisons?

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*
- Which of these comparisons were unnecessary?
  - How can you search with fewer comparisons?

More generally: How can we use results of *previous comparisons* to avoid making unnecessary comparisons in the future?

For Next Time

Consider How could we improve upon BRUTEFORCEMATCH

- How can we use information about *previous matches* in order to avoid doing some *future checks*?

Scratch Notes