Divide and Conquer Algorithms

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Definition

Divide and conquer algorithms are a class of algorithms that solve a problem by

- breaking it into smaller subproblems,
- solving the subproblems recursively,
- and then combining the solutions to the subproblems to form a solution to the original problem.

Problems that can be solved in this manner are typically highly parallelizable.

Definition

A divide and conquer method is split into three steps:

1. Divide the problem into smaller subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine the solutions to the subproblems to form a solution to the original problem.

Definition

Their runtime can be characterized by the recurrence relation $T(n)$.

A recurrence $T(n)$ is *algorithmic* if, for every sufficiently large *threshold* constant $n_0 > 0$, the following two properties hold:

1. For all $n \leq n_0$, the recurrence defines the running time of a constant-size input.
2. For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive calls.

Definition

The algorithm must output a solution in finite time.

If the second property doesn't hold, the algorithm is not correct – **it may end up in an infinite loop.**

Definition

"Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic." - Cormen et al.

Definition

This assumption means that the algorithm is correct and terminates in finite time: **the base case is implied.**

The base case is less important for analysis than the recursive case.

For example, your base case might work with 100 elements, and that would still be $\Theta(1)$ because it is a constant.

Definition

It is common to break up each subproblem uniformly, but it is not always the best way to do it.

For example, an application such as matrix multiplication is typically broken up uniformly since there is no spatial or temporal relationship to consider.

Definition

Algorithms for image processing may have input values that are locally correlated.

It may be better to break up the input in a way that preserves this correlation.

Example: Merge Sort

Merge Sort

Merge sort is a classic example of a divide and conquer algorithm.

It works by dividing the input array into two halves, sorting each half recursively, and then merging the two sorted halves.

Divide

The divide step takes an input subarray $A[p : r]$ and computes a midpoint q before partitioning it into two subarrays $A[p : q]$ and $A[q + 1 : r]$.

These subarrays will be sorted recursively until the base case is reached.

Conquer

The conquer step recursively sorts the two subarrays $A[p : q]$ and $A[q + 1 : r]$.

If the base case is such that the input array has only one element, the array is already sorted.

Combine

The combine step merges the two sorted subarrays to produce the final sorted array.

Merge Sort

```
def merge_sort(A):
    if len(A) <= 1:
        # Conquer -- base case
        return A

    # Divide Step
    mid = len(A) // 2
    left = merge_sort(A[:mid])
    right = merge_sort(A[mid:])

    # Combine Step
    return merge(left, right)
```

Merge Sort

```
def merge(left, right):
    result = []
    i, j = 0, 0

    # Merge the two subarrays
    while (i < len(left)) and (j < len(right)):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add the remaining elements to the final array
    result += left[i:]
    result += right[j:]
    return result
```

Merge Sort: Analysis

When analyzing the running time of a divide and conquer algorithm, it is safe to assume that the base case runs in constant time.

The focus of the analysis should be on the **recurrence equation**.

Merge Sort: Analysis

- We originally have a problem of size n.
- Divide the problem into 2 subproblems of size $n/2$.
- The recurrence is $T(n) = 2T(n/2)$.
- This continues for as long as the base case is not reached.

Merge Sort: Analysis

Factor in the time it takes for the divide and combine steps.

- These can be represented as $D(n)$ and $C(n)$, respectively.
- $T(n) = 2T(n/2) + D(n) + C(n)$ when $n >= n_0$, where n_0 is the base case.

Merge Sort: Analysis

- Divide step is $D(n) = \Theta(1)$ since all it does is compute the midpoint.

Merge Sort: Analysis

- Divide step is $D(n) = \Theta(1)$ since all it does is compute the midpoint.
- Conquer step is the recurrence $T(n) = 2T(n/2)$.

Merge Sort: Analysis

- Divide step is $D(n) = \Theta(1)$ since all it does is compute the midpoint.
- Conquer step is the recurrence $T(n) = 2T(n/2)$.
- Combine step is $C(n) = \Theta(n)$ since it takes linear time to merge the two subarrays.

Merge Sort: Analysis

- Divide step is $D(n) = \Theta(1)$ since all it does is compute the midpoint.
- Conquer step is the recurrence $T(n) = 2T(n/2)$.
- Combine step is $C(n) = \Theta(n)$ since it takes linear time to merge the two subarrays.
- The worst-case running time of merge sort is $T(n) = 2T(n/2) + \Theta(n)$.

Merge Sort: Analysis

Not every problem will have a recurrence of $2T(n/2)$.

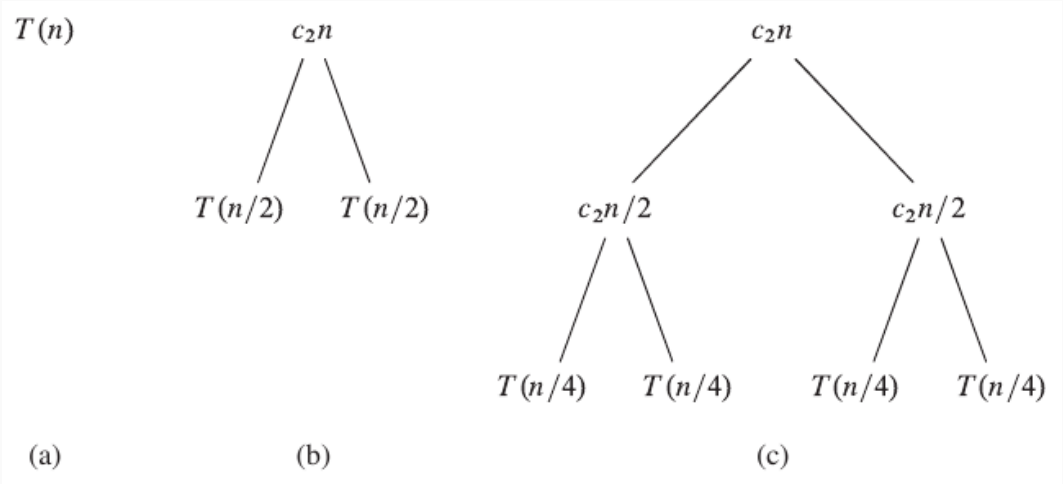
We can generalize this to $aT(n/b)$, where a is the number of subproblems and b is the size of the subproblems.

Merge Sort: Analysis

The recurrence does not tell us the asymptotic upper bound.

Let's look at a recursion tree of the execution of merge sort.

Merge Sort: Analysis



Recursion tree for merge sort.

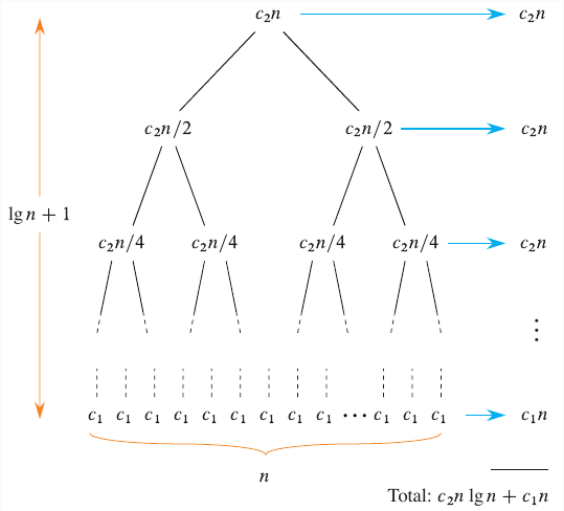
Merge Sort: Analysis

- The root of the tree represents the original problem of size n in (a).
- In (b), the divide step splits the problem into two problems of size $n/2$.
- The cost of this step is indicated by $c_2 n$.
- c_2 represents the constant cost per element for dividing and combining.

Merge Sort: Analysis

- The combine step is dependent on the size of the subproblems, so the cost is $c_2 n$.
- Subfigure (c) shows a third split, where each new subproblem has size $n/4$.
- This would continue recursively until the base case is reached.

Merge Sort: Analysis



Recursion tree for merge sort.

Merge Sort: Analysis

Finalizing the analysis

- The upper bound for each level of the tree is $c_2 n$.

Merge Sort: Analysis

Finalizing the analysis

- The upper bound for each level of the tree is $c_2 n$.
- The height of a binary tree is $\log_b n$.

Merge Sort: Analysis

Finalizing the analysis

- The upper bound for each level of the tree is $c_2 n$.
- The height of a binary tree is $\log_b n$.
- The total cost of the tree is the sum of the costs at each level.

Merge Sort: Analysis

Finalizing the analysis

- The upper bound for each level of the tree is $c_2 n$.
- The height of a binary tree is $\log_b n$.
- The total cost of the tree is the sum of the costs at each level.
- In this case, the cost is $c_2 n \log n + c_1 n$, where the last $c1_n$ comes from the base case.

Merge Sort: Analysis

Finalizing the analysis

- The upper bound for each level of the tree is $c_2 n$.
- The height of a binary tree is $\log_b n$.
- The total cost of the tree is the sum of the costs at each level.
- In this case, the cost is $c_2 n \log n + c_1 n$, where the last $c1_n$ comes from the base case.
- The first term is the dominating factor in the running time, so the running time of merge sort is $\Theta(n \log n)$.

The Substitution Method

Substitution Method

The **substitution method** is a technique for solving recurrences. It works in two steps:

- Guess the solution
- Use mathematical induction to verify the guess

Substitution Method

Let's start with an example: **The Tower of Hanoi**.

In this classic puzzle, we have three pegs and a number of disks of different sizes which can slide onto any peg.

The puzzle starts with the disks in a neat stack in ascending order of size on one peg, with the smallest disk on top.

Substitution Method

Substitution Method

The objective is to move the entire stack to another peg, obeying the following rules:

1. Only one disk can be moved at a time
2. Each move consists of taking the top disk from one of the stacks and placing it on top of another stack
3. No disk may be placed on top of a smaller disk

Substitution Method

An algorithm to solve the puzzle goes like this:

1. Move $n-1$ disks from peg 1 to peg 2 using peg 3 as a temporary holding area
2. Move the nth disk from peg 1 to peg 3
3. Move the $n-1$ disks from peg 2 to peg 3 using peg 1 as a temporary holding area

Substitution Method

The number of moves required to solve the Tower of Hanoi puzzle is given by the recurrence relation $T(n) = 2T(n-1) + 1$ with the initial condition $T(1) = 1$.

We can solve this recurrence using the substitution method.

Substitution Method

What is our guess?

Substitution Method

What is our guess?

Our hypothesis might be that $T(n) \leq c2^n - 1$ for all $n \geq n_0$, where $c > 0$ and $n_0 > 0$.

Substitution Method

For the base case, we have $T(1) = c * 2^1 - 1 = 1$, so $c = 1$.

Now we need to show that $T(n) \leq c2^n - 1$ for all $n \geq n_0$.

Assume that $T(k) \leq c2^k - 1$ for all $k < n$.

Plug in the guess

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

What if we made a bad guess?

Let's try $T(n) \leq cn$ for all $n \geq n_0$.

- We have $T(1) = c = 1$, so $c = 1$.
- Now we need to show that $T(n) \leq cn$ for all $n \geq n_0$.
- Assume that $T(k) \leq ck$ for all $k < n$.

Substitution Method

$$
\begin{aligned}
T(n) &\leq 2T(n-1)+1 \\
&\leq 2c(n-1)+1 \\
&= 2cn-2c+1
\end{aligned}
$$

This does not work because $2cn-2c+1 > cn$ for all $c > 1$.

Substitution Method

What about $T(n) \leq c2^n$?

Class Exercise

Class Exercise: Determine an asymptotic upper bound for

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n).$$

Class Exercise

Guess: $T(n) = O(n \lg n)$

Class Exercise

Guess: $T(n) = O(n \lg n)$

Inductive hypothesis: $T(n) \leq cn \lg n$ for all $n \geq n_0$.

Class Exercise

Guess: $T(n) = O(n \lg n)$

Inductive hypothesis: $T(n) \leq cn \lg n$ for all $n \geq n_0$.

Inductive step: Assume $T(n) \leq cn \lg n$ for all $n_0 \leq k < n$. For $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$, it holds when $n \geq 2$.

Proof

$$
\begin{aligned}
T(n) &\leq 2T(\lfloor n/2 \rfloor) + \Theta(n) \\
&\leq 2c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - 2cn \lg 2 + \Theta(n) \\
&= cn \lg n - 2cn + \Theta(n) \\
&\leq cn \lg n
\end{aligned}
$$

Recursion-Tree Method

Recursion-Tree Method

Visualizing the characteristics of an algorithm is a great way to build intuition about its runtime.

Although it can be used to prove a recurrence, it is often a good jumping off point for the substitution method.

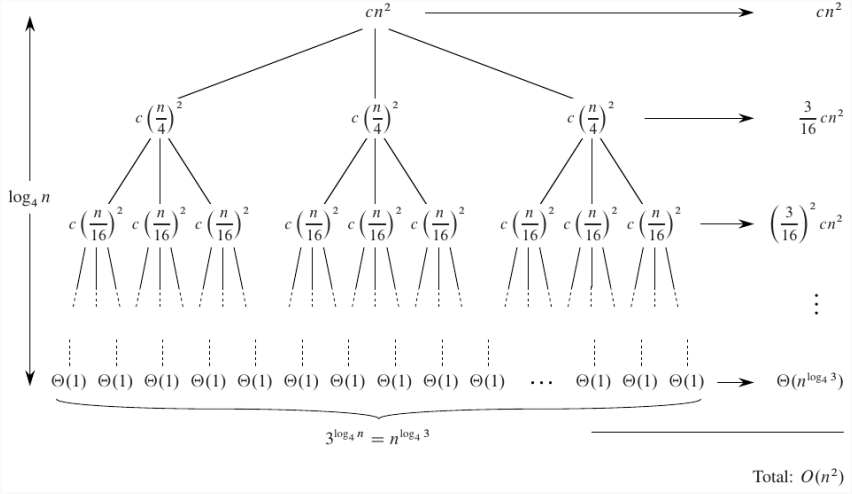
Recursion-Tree Method

Consider the recurrence $T(n) = 3T(n/4) + \Theta(n^2)$.

We start by describing $\Theta(n^2) = cn^2$, where the constant $c > 0$ serves as an upper-bound constant.

It reflects the amount of work done at each level of the recursion tree.

Recursion-Tree Method



Recursion tree for $T(n) = 3T(n/4) + \Theta(n^2)$.

Tree Analysis

- Observe the pattern in the cost at depth i.
- Thus, the subproblem size at depth i is $n/4^i$.
- Each level of increasing depth has 3 times as many nodes as the previous.
- With the exception of the leaves, the cost for each level is $(\frac{3}{16})^i cn^2$.

Tree Analysis

The subproblem size at depth i is $n/4^i$.

Tree Analysis

The subproblem size at depth i is $n/4^i$.

When does the base case occur?

Tree Analysis

The subproblem size at depth i is $n/4^i$.

When does the base case occur?

The base case occurs when $n/4^i = 1$, or $i = \log_4 n$.

Recursion-Tree Method

The total cost of the leaves is based on the number of leaves, which is $3^{\log_4 n}$ since each level has 3^i nodes and the depth is $\log_4 n$.

Using the identity $a^{\log_b c} = c^{\log_b a}$, we can simplify the leaves to $n^{\log_4 3}$.

The total cost of the leaves is $\Theta(n^{\log_4 3})$.

Review: Geometric Series

For real numbers a and r where $r \neq 1$,

$$\sum_{i=0}^{k} ar^i = a\frac{r^{k+1} - 1}{r - 1}.$$

If $|r| < 1$, then

$$\sum_{i=0}^{\infty} ar^i = \frac{a}{1 - r}.$$

Recursion-Tree Method

The last step is to add up the costs over all levels:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{cn^2}{1-\frac{3}{16}} + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= \Theta(n^2).
\end{aligned}
$$

Verifying the Solution

Let's verify that this recurrence is bounded above by $O(n^2)$ using the substitution method.

Here we show that $T(n) \leq dn^2$ for a constant $d > 0$. The previous constant $c > 0$ is reused to describe the cost at each level of the recursion tree.

Verifying the Solution

$$
\begin{aligned}
T(n) &\leq 3T(n/4) + cn^2 \\
&\leq 3(d(n/4)^2) + cn^2 \\
&= \frac{3}{16}dn^2 + cn^2 \\
&\leq dn^2 \text{ if } d \geq \frac{16}{13}c.
\end{aligned}
$$

Class Exercise

Draw a recursion tree for $T(n) = 2T(n/5) + cn$.

Example: Multiplying Square Matrices

Example: Multiplying Square Matrices

Matrix multiplication can be divided into subproblems thanks to the properties of linear combinations.

A divide and conquer algorithm hinges on a base case using the definition of matrix multiplication.

Example: Multiplying Square Matrices

```
def square_matrix_multiply(A, B):
    n = len(A)
    C = [[0 for _ in range(n)] for _ in range(n)] # O(n^2)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Example: Multiplying Square Matrices

- The matrix is split into block matrices of size $n/2$.
- Each submatrix can be multiplied with the corresponding submatrix of the other matrix.
- The resulting submatrices can be added together to form the final matrix.

Example: Multiplying Square Matrices

- Base case is $n = 1$ where only a single addition and multiplication are performed: $T(1) = \Theta(1)$.
- For $n > 1$, the recursive algorithm starts by splitting into 8 subproblems of size $n/2$.
- There are 8 subproblems because there are 4 submatrices in each matrix, and each submatrix is multiplied with the corresponding submatrix in the other matrix.

Example: Multiplying Square Matrices

Building the recurrence relation

- Each recursive call contributes $T(n/2)$ to the running time.

Example: Multiplying Square Matrices

Building the recurrence relation

- Each recursive call contributes $T(n/2)$ to the running time.
- There are 8 recursive calls, so the total running time is $8T(n/2) + \Theta(n^2)$.

Example: Multiplying Square Matrices

Building the recurrence relation

- Each recursive call contributes $T(n/2)$ to the running time.
- There are 8 recursive calls, so the total running time is $8T(n/2) + \Theta(n^2)$.
- There is no need to implement a combine step since the matrix is updated in place.

Example: Multiplying Square Matrices

Building the recurrence relation

- Each recursive call contributes $T(n/2)$ to the running time.
- There are 8 recursive calls, so the total running time is $8T(n/2) + \Theta(n^2)$.
- There is no need to implement a combine step since the matrix is updated in place.
- The final running time is $T(n) = 8T(n/2) + \Theta(1)$ **for the recursive portion**.

Example: Multiplying Square Matrices

Consider 4×4 matrix multiplication AB...

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Example: Multiplying Square Matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where each submatrix is of size 2×2.

These matrices are already partitioned.

Example: Multiplying Square Matrices

They currently don't meet the base case, so 8 recursive calls are made which compute the following products:

1. $A_{11}B_{11}$
2. $A_{12}B_{21}$
3. $A_{11}B_{12}$
4. $A_{12}B_{22}$
5. $A_{21}B_{11}$
6. $A_{22}B_{21}$
7. $A_{21}B_{12}$
8. $A_{22}B_{22}$

Example: Multiplying Square Matrices

In the first recursive call, the 2×2 matrices are partitioned into 4 1×1 matrices, or scalars.

The base case is reached, and the product is computed.

The same process is repeated for the other 7 recursive calls and the final matrix is formed by adding the products together.

Example: Multiplying Square Matrices

```
def partition(A):
    n = len(A)
    mid = n // 2
    A11 = [row[:mid] for row in A[:mid]]
    A12 = [row[mid:] for row in A[:mid]]
    A21 = [row[:mid] for row in A[mid:]]
    A22 = [row[mid:] for row in A[mid:]]
    return A11, A12, A21, A22
```

Example: Multiplying Square Matrices

```
def matrix_multiply_recursive(A, B, C, n):
    if n == 1:
        C[0][0] += A[0][0] * B[0][0]
    else:
        # Partition the matrices
        A11, A12, A21, A22 = partition(A)
        B11, B12, B21, B22 = partition(B)
        C11, C12, C21, C22 = partition(C)
```

Example: Multiplying Square Matrices

```
# Recursively compute the products
matrix_multiply_recursive(A11, B11, C11, n/2)
matrix_multiply_recursive(A12, B21, C11, n/2)
matrix_multiply_recursive(A11, B12, C11, n/2)
matrix_multiply_recursive(A12, B22, C11, n/2)
matrix_multiply_recursive(A21, B11, C11, n/2)
matrix_multiply_recursive(A22, B21, C11, n/2)
matrix_multiply_recursive(A21, B12, C11, n/2)
matrix_multiply_recursive(A22, B22, C11, n/2)
```

Example: Multiplying Square Matrices

Class Exercise: Recursion Tree for Matrix Multiplication

- $T(n) = 8T(n/2) + \Theta(n^2)$ worst-case divide step
- $T(n) = 8T(n/2) + \Theta(1)$ best-case divide step

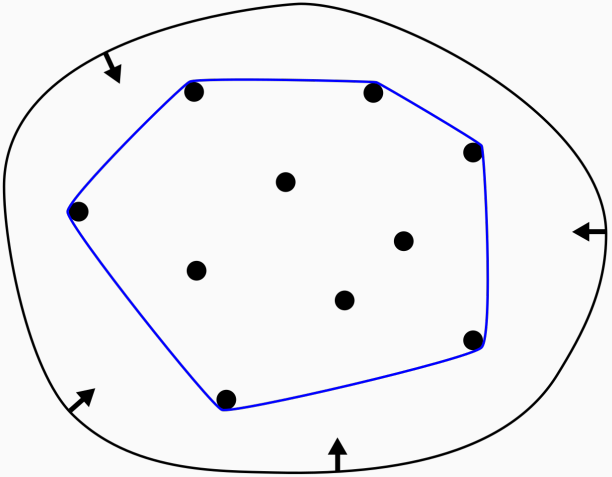
Example: Convex Hull

Convex Hull

Given n points in plane, the convex hull is the smallest convex polygon that contains all the points.

- No two points have the same x or y coordinate.
- Sequence of points on boundary in clockwise order as doubly linked list.

Convex Hull



The convex hull problem (Wikipedia).

Convex Hull

Naive Solution?

Convex Hull

Naive Solution?

- Draw lines between each pair of points.
- If all other points are on the same side of the line, the line is part of the convex hull.
- This is $\Theta(n^3)$.

Convex Hull: Divide and Conquer

- Sort the points by x coordinate.
- Split into two halves by x.
- Recursively find the convex hull of each half.
- Merge the two convex hulls.

Convex Hull: Divide and Conquer

The **divide** step is straightforward. Simply split the points into two halves by x.

The **merge** step will be the most complex part of the algorithm.

Convex Hull: Divide and Conquer

What is the base case of our problem?

Convex Hull: Divide and Conquer

What is the base case of our problem?

If there are only 3 points, the convex hull is the triangle formed by the 3 points.

Convex Hull: Divide and Conquer

There are several algorithms to merge the hulls together.

Let's take a look at the **gift wrapping** algorithm.

Convex Hull: Divide and Conquer

Start at the rightmost point of the left convex hull and the leftmost point of the right convex hull.

Move the right finger clockwise and the left finger counterclockwise until the upper tangent is found.

Repeat for the lower tangent.

Convex Hull: Divide and Conquer

With both the upper and lower tangents found, the two convex hulls can be merged together.

Remove the points between the two tangents and add the points from the other hull.

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

- Imagine a vertical line splitting the two hulls.

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

- Imagine a vertical line splitting the two hulls.
- The upper tangent is the line with the greatest intercept across the vertical line.

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

- Imagine a vertical line splitting the two hulls.
- The upper tangent is the line with the greatest intercept across the vertical line.
- The lower tangent is the line with the smallest intercept across the vertical line.

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

Convex Hull: Divide and Conquer

How do we determine the upper and lower tangents?

In practice, we will use the **orientation test**.

Orientation Test

The orientation test is a technique from computational geometry which determines the orientation of three points.

It will tell us if a third point lies below or above a given line segment.

We can use it to determine if a point is part of the convex hull.

Orientation Test

Given three points p, q, and r, the orientation is determined by the sign of the cross product of the vectors $\overrightarrow{pq}$ and $\overrightarrow{pr}$.

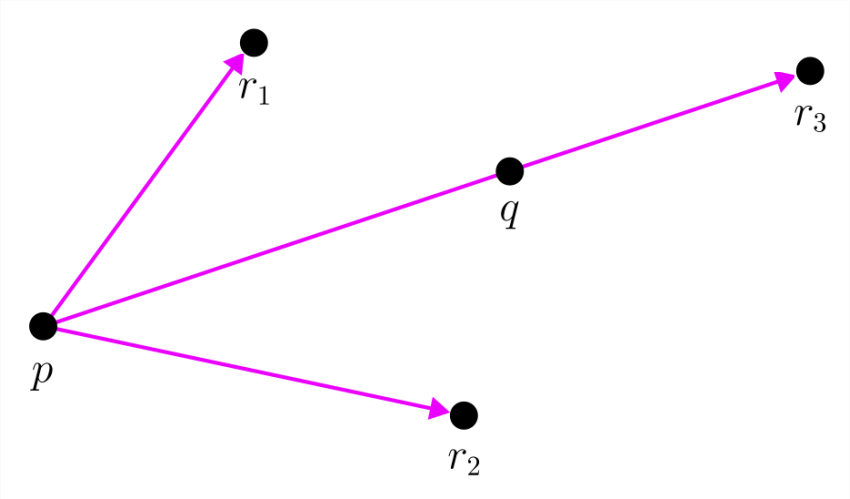
- If positive, the orientation is clockwise.
- If negative, the orientation is counterclockwise.
- If zero, the points are collinear.

Orientation Test

Given three points p, q, and r, the orientation is determined by the sign of the cross product of the vectors $\overrightarrow{pq}$ and $\overrightarrow{pr}$.

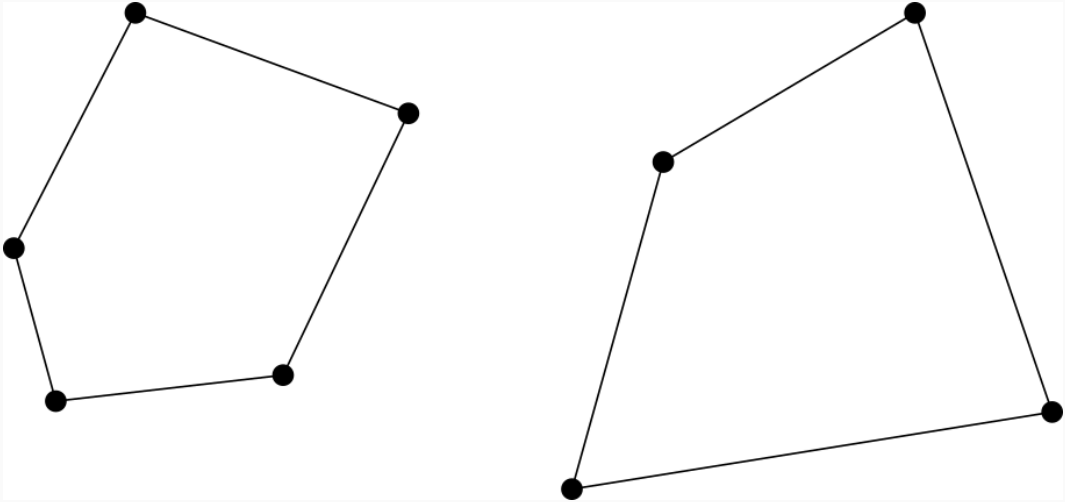
- If positive, the orientation is clockwise.
- If negative, the orientation is counterclockwise.
- If zero, the points are collinear.

Orientation Test



Orientation test.

Orientation Test



Merge example.

Convex Hull: Analysis

Class Exercise: Analyze the running time of the convex hull algorithm.

To get started...

- What is the recurrence?
- What is the complexity of the merge operation?