General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Examples: $T(n) = 2T(n/2) + C \Rightarrow T(n) \in ?$
What if a = 1, 4; what term dominates?

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in ?$
What if a = 4, 8; what term dominates?

$T(n) = 2T(n/2) + n^2 \Rightarrow T(n) \in ?$
What if a = 4, 8; what term dominates?

General Divide-and-Conquer Recurrence

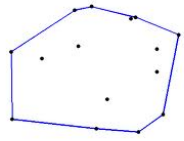
$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$ or $\log_b(a) < d$, $T(n) \in \Theta(n^d)$
If $a = b^d$ or $\log_b(a) = d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$ or $\log_b(a) > d$, $T(n) \in \Theta(n^{\log_b a})$

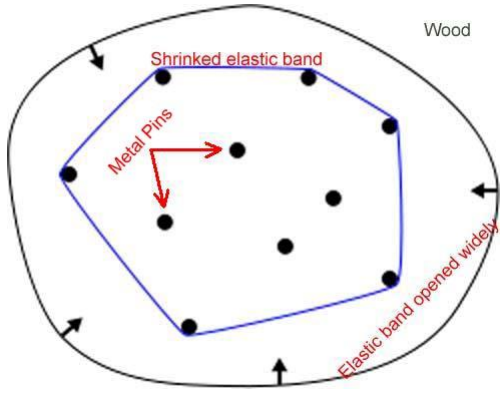
Note: The same results hold with O instead of Θ.

Convex Hull Problem: given a set of points, find the smallest convex set containing the points, Convex Hull

- A convex combination of two distinct points is any point on the line segment between them.
- Convex set: A set of points in the plane is called convex if, for any two points p and q in the set, the entire line segment from p to q including the endpoints belongs to the set.
- Convex hull of a set S is the smallest convex set that includes S
- To "solve" the convex hull problem we will find the extreme points of the convex set – that is the corners of the convex hull.



Convex Hull – Physical determination

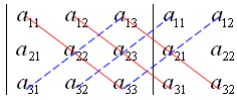


Some necessary computational facts

Two key steps: Determine

- Where a point, p_3, is to the left or right of a line (with direction), e.g. line from p_1 to p_2 .
- The distance of a point p from a line.

- Great news – both of these are solved by one calculation
  - Det of $\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$
  - Det > 0, then p_3 is to the left of $\overrightarrow{p_1p_2}$; < 0, to the right ; = 0, on the line
  - Finally normalizing by the length of the segment $\overrightarrow{p_1p_2}$ gives the distance



Quickhull Algorithm

- Convex hull: smallest convex set that includes given points
- Assume points are sorted by x-coordinate values
- Identify extreme points P_1 and P_2 (leftmost and rightmost)
- Compute upper hull recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute lower hull in a similar manner

Quickhull Algorithm

Input = a set S of n points
Assume that there are at least 2 points in the input set S of points

QuickHull (S)
// Find convex hull from the set S of n points
Convex Hull := {}
Find left and right most points, say A & B, and add A & B to convex hull
Segment AB divides the remaining (n-2) points into 2 groups S_1 and S_2
where S_1 are points in S that are on the left side of the oriented line from A to B,
and S_2 are points in S that are on the left side of the oriented line from B to A
FindHull (S_1, A, B)
FindHull (S_2, B, A)

Quickhull Algorithm

FindHull (S_k, P, Q)
// Find points on convex hull from the set S_k of points
// that are on the left side of the oriented line from P to Q
If S_k has no point then return
From the given set of points in S_k, find farthest point, say C, from segment PQ
Add point C to convex hull at the location between P and Q
Three points P, Q, and C partition the remaining points of S_k into 3 subsets:
S_0 are points inside triangle PCQ,
S_1 are points on the left side of the oriented line from P to C, and
S_2 are points on the left side of the oriented line from C to Q.
FindHull(S_1, P, C)
FindHull(S_2, C, Q)

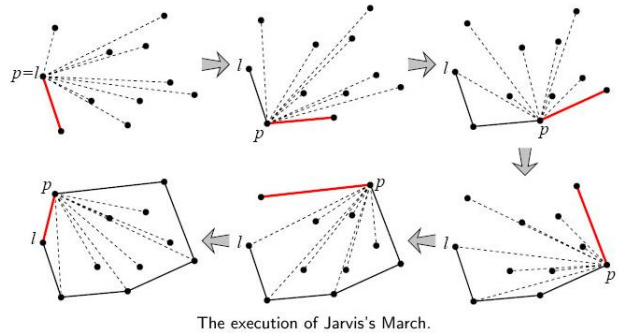
Output = convex hull

Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x-coordinate value, this can be accomplished in O(n log n) time
- Several O(n log n) algorithms for convex hull are known

Convex Hull Problem: Jarvis March (Wrapping algorithm)

Algorithm finds the points on the convex hull in the order in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many. Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. etc. (O(nh)) h: #pts in CH)



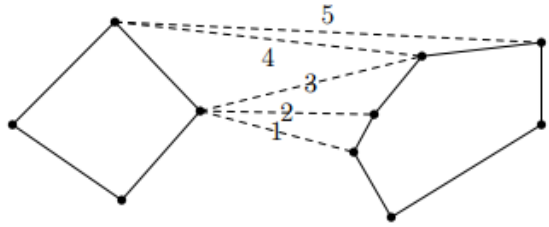
The execution of Jarvis's March.

Convex Hull Problem: (Pure) Divide and Conquer

Divide and conquer

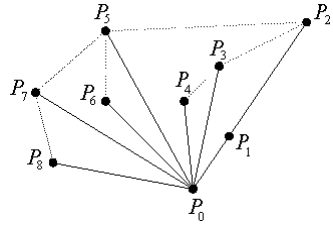
1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Combine! (march up/down until upper/lower tangent)



Convex Hull Problem: Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order. Let x_0 be the leftmost point and number the remaining points by angle from x_0 going counterclockwise: x_1; x_2; : : : ; x_{n-1}. Let $x_n = x_0$, the chosen point.



Greedy Technique

- Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:
  - *feasible*
  - *locally optimal*
  - *irrevocable*
- For some problems, yields an optimal solution for every instance.
- For some problems where it does not yield an optimal solution it can be useful for fast approximations

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Total Weighted Time to Completion Problem

Total Weighted Time to Completion Problem: Given a set of jobs: job_1, job_2, …, job_n characterized by their lengths: l_1, l_2, … , l_n and their weights: w_1, w_2, … , w_n

- Output a schedule that minimizes the total weighted time to completion.
  - Schedule: An ordering of the jobs to be serviced by the resource
  - Completion time: C_i, of job l_i is the sum of the l_j's up to and including l_i
- Total Weighted Time to Completion = $\sum_{i=1 \text{ to } n} w_i C_i$

Question?

- Given 3 jobs job_1, job_2, job_3 where the respective lengths are 1, 2, 3. What are the Completion times of the three jobs if done in the order given?
  - 1, 2, 3
  - 3, 5, 6
  - 1, 3, 6
  - 1, 4, 6
- If the weights are respectively 3, 2, 1. What is the Total Weighted Time to Completion for this ordering?
- Note: There are n! orderings.

Problem (TWTC)

Finding a Greedy algorithm that solves this problem:
Focus on process

- Goal: Find a sequence of a set of jobs that **minimizes** the total weighted time to completion.
- There may be many candidate greedy algorithms!
- How do we find a greedy algorithm that works.
  - Look at special cases
  - Generate candidate greedy algorithms
  - Narrow down to single (if possible) candidate, usually by looking at simple cases to eliminate candidates, then prove correctness. (It may be that no greedy algorithm works.)
  -

Discovery of candidates

- Explore special cases:
  - Equal lengths
  - Equal weights
- What order do you think you should schedule jobs?
  - Larger weights first; shorter lengths first
  - Smaller weights first; shorter lengths first
  - Larger weights first; longer lengths first
  - Smaller weights first; longer lengths first
- Hint: Look at small examples to test your ideas

What are reasonable candidates?

- How can we combine the information we have into a greedy algorithm
- Two simple options, order the jobs by either:
  - Difference: $w_i - l_i$ Ratio: w_i / l_i
- How to rule one of these out?
  - Find, if possible, an example where the two different algorithms give two different answers.
  - Clearly the larger solution for the example is not optimal.
  - This rules out that algorithm but does not prove anything about the algorithm that gives the smaller solution.

TWTC example problem

Consider 2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

	$W_i - L_i$	W_i / L_i
Job_1	4	5
Job_2	8	3
Ordering	Job_2 , Job_1	Job_1 , Job_2
TWTC	48+25 = 73	5+60 =65

- $W_i - L_i$ does not always yield the optimal (smallest) TWTC.

Proving correctness: exchange argument

Plan: Assume the greedy algorithm does not produce the optimal solution.

- Assume the w/l ratios are all distinct so there is only one possible ordering given by the algorithm.
- Assume that the jobs are labeled according to the decreasing w/l ratio, e.g. w_1 / l_1 is the largest ratio, w_n / l_n is smallest
- Assume that there is an ordering $\sum^*$ that is the optimal solution but is different from $\sum$ the ordering given by the algorithm

Now show that $\sum^*$ that is not the optimal solution.

This contradicts our assumption that it is optimal and different from $\sum$. Hence the optimal solution must be $\sum$.

Proving correctness: exchange argument details

the jobs are labeled according to the decreasing ratios

If $\sum^*$ is not $\sum$, it must have two consecutive jobs where i > j but i comes before j in $\sum^*$. Why??

Order of jobs in $\sum^*$

Jobs before i	Job i	Job j	Jobs after j

If switch jobs i and j, what happens to the completion times of jobs?

Proving correctness: exchange argument details

Jobs before i	Job i	Job j	Jobs after j

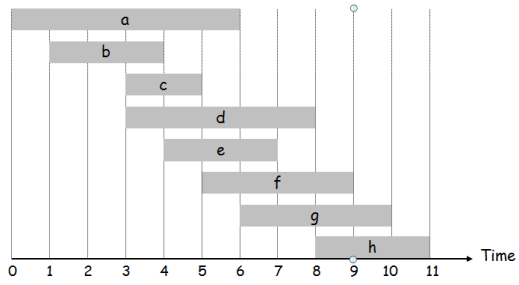
If switch jobs i and j, what happens to the completion times of jobs?

$k \neq i, j$	i	j
– unaffected	up	down

- Now what does switching i and j do to the total cost?
  - Since i > j we know that $w_i / l_i < w_j / l_j$ so
  - $w_i * l_j < w_j * l_j$ BUT
  - the ordering after making the switch has a lower total weighted time to completion! CONTRADICTION that $\sum^*$ was the minimum total weighted time to completion

Interval Scheduling

- Interval scheduling.
  - Job j starts at sj and finishes at fj.
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.
  - [Earliest start time] Consider jobs in ascending order of start time sj.
  - [Earliest finish time] Consider jobs in ascending order of finish time fj.
  - [Shortest interval] Consider jobs in ascending order of interval length fj - sj.
  - [Fewest conflicts] For each job, count the number of conflicting jobs cj. Schedule in ascending order of conflicts cj.

Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

Interval Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken. (Earliest finish time first)

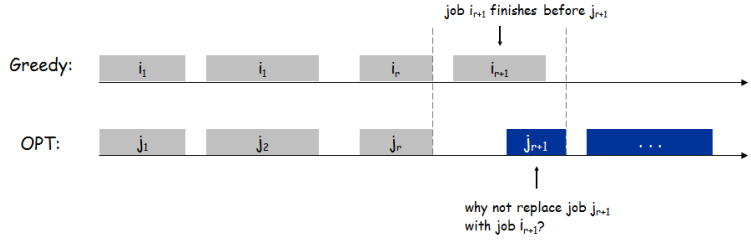
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
                    jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

- Implementation. O(n log n).
  - Remember job j* that was added last to A.
  - Job j is compatible with A if sj ≥ fj*.

Interval Scheduling – Stay Ahead proofs
Proof of correctness

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.