What you should learn from CS341

- Main Idea:
  - Given a problem, how do we design an efficient algorithm that solves the problem.
- What you should learn:
  - Some good algorithms for basic problems
  - Paradigms or ways to solve problems
  - Proving algorithm correctness
  - Assessing algorithm efficiency

Course Overview

- **Introduction:**
  - An example of how designing a better algorithm can help build a faster program.
  - Review: Notion of problem, algorithm, time complexity and asymptotic notation.

- **Algorithm design techniques:**
  - Every problem needs to be approached individually.
    - There is no magic method that can be used to design efficient algorithms.
    - It is a creative area of endeavour but the more often you design efficient algorithms the better you will be at algorithm design.
  - There are a few standard techniques that can help:
    - Greedy algorithms, Divide&Conquer, Dynamic programming.

Course Overview (cont.)

- **Graph Algorithms:**
  - Graphs can be used to represent many real-life problems.
  - It is very useful to know how to solve standard problems on graphs efficiently.
  - Also, graphs provide a good application area for methods presented in previous sections.
- **Intractability and Undecidability**

Why would you ever use this stuff?

- **Algorithm**: a way of solving a problem
- **Program**: an implementation of an algorithm
- Design choices:
  - For any problem there are many algorithms.
  - For any algorithm there are many implementations.
- It is useful to do as much assessment as possible before implementation.
  - Think before typing (not type then debug then suffer).

Problem Solving in the Workplace

- Come across a problem in an application area.
  - The problem may be “disguised” and not readily seen as a “classic” problem.
- You should:
  - Try various paradigms.
  - Search the literature.
  - Be able to justify your approach at an early stage.

What makes an algorithm “Good”?

- It is **correct**
  - It always terminates with the right answer.
- It is **efficient**
  - What does this mean?
    - We need a model of computation to be precise about the concept of efficiency.

  - We will get back to the idea of a model in a later section.
  - For now we continue with a motivational example.

Bentley's Problem

- Given $A[1..n]$, find $\max_{1 \le i \le j \le n} \sum_{k=i}^{j} A[k]$

  or return 0 if all elements of the array are negative.
  - Used by Bentley in his book "Programming Pearls" to illustrate why algorithm design and analysis is important to programmers.
  - Example: **`31 -41 59 26 -53 58 97 -93 -23 84`**

    `                ^^^^^^^^^^^^^^^^^`
    - This subarray has the maximum sum of **`187`**.
- There is an obvious $\Theta(n^3)$ algorithm.
- Clever programmers can get this down to $\Theta(n^2)$.

Bentley’s Problem (Solution 1)

- Evaluate all possible subarrays and select the one with the largest sum.

```
max := 0;
for i := 1 to n do
   for j := i to n do
      // compute sum of subarray A[i]..A[j]
      sum := 0;
      for k := i to j do
         sum := sum + A[k];
      // compare to maximum
      if sum > max then max := sum;
```

Bentley's Problem (Solution 1 Running Time)

- **Recall:**
  - "Θ notation" for measuring how running time grows with the size of the input.
- **Informally:**
  - Running time of a problem with input of size n is $\Theta(f(n))$ if it grows proportionally to $f(n)$.
- **Time:**
  - In this case running time is $\Theta(n^3)$
- Question: Can we do better?

Bentley's Problem (Solution 2a)

- Solution 1 computes the sum by adding up all entries in the subarray with end points determined by the two outer loops. We can avoid this.

```
max := 0;
for i := 1 to n do
   sum := 0;
   for j := i to n do
      sum := sum + A[j];
         // sum is now sum of subarray A[i]..A[j]
         // Compare to maximum
         if sum > max then max := sum;
```

  – Running time is now $\Theta(n^2)$.

Bentley's Problem (Solution 2b)

- We can compute the sum in constant time if we do a little bit of pre-computation:
  - Let B[i] be the sum of A[1] + … + A[i].
  - Then A[i] + … + A[j] = B[j] - B[i - 1].

```
// precompute B[i] = A[1] + ... + A[i]
B[0] := 0;
for i := 1 to n do
   B[i] := B[i-1] + A[i];
max := 0;
for i := 1 to n do
   for j := i to n do
      // Compare to maximum
      if B[j] - B[i-1] > max then
         max := B[j] - B[i-1];
```

  - Running time is still $\Theta(n^2)$.

Bentley’s Problem (Solution 3): Divide-and-Conquer

- Recall MergeSort:
  - To sort an array:
    - Divide an array into two equally-sized parts.
    - Sort each part separately.
    - Solution is obtained by “merging" the smaller solutions.

Bentley's Problem (Solution 3): Divide-and-Conquer

- Divide-and-Conquer can also be used here:
  - Divide an array into two equally-sized parts.
  - Our solution must either be entirely in the left part, or entirely in the right part, or it must be crossing the partition line. Therefore:
- Find the maximum subarray for left part (maxL) and right part (maxR) (done by recursive call).
- Find the maximum subarray "going over the middle partition line" (maxM).
  - This can be done in linear time $\Theta(n)$.

- The solution is max{maxL, maxR, maxM}.
- It can be shown that the running time is $\Theta(n \log n)$.

Bentley's Problem (Solution 3): Divide-and-Conquer

```
recursive maxsum(low,hi)
  if low > hi return 0;    // zero element vector
  if low = hi return max(0, A[low]); // 1 element vec
  mid := (low + hi)/2;
  // Find max from the partition down to the left.
  leftmax := sum := 0;
  for i := mid downto low
     sum := sum + A[i];
     leftmax := max(leftmax,sum);
  rightmax := sum := 0;   //Now do same for right
  for i := mid + 1 to hi
     sum := sum + A[i];
     rightmax := max(rightmax,sum);
  return(max{leftmax + rightmax, maxsum(low,mid),
                          maxsum(mid + 1, hi)});
```

Running time?

Bentley's Problem:
Can we do better than $\Theta(n \log n)$?

- Let *maxsol*(i) be the maximum solution for array $A[1..i]$.
- What is the relationship between *maxsol(i)* and *maxsol*(i-1)?
- Note: *maxsol*(i) = max{*maxsol*(i-1), *tail*(i)} where *tail*(i) is the maximum solution ending at position i.
- That is: *tail*(i) = max{*tail*(i-1) + $A[i]$, 0}

Bentley's Problem (Solution 4)

- Bentley's problem can be done in $\Theta(n)$ time.

```
maxsol := 0;    tail := 0;
for i := 1 to n do
   tail := max(tail + A[i], 0);
   maxsol := max(maxsol, tail);
```

Bentley's Problem: Time Comparisons

- Consider solutions implemented in C.
  - Some of the values were measured (on a Pentium II), some of them were estimated from other measurements.
  - ε represents a time under 0.01s
  - Solution 0 is supposedly an exponential-time solution.
    - We have contrived this unrealistic "solution" just for comparison with the others so that you can see how awful an exponential time solution would be.

Bentley's Problem: Time Comparisons

		Sol.4	Sol.3	Sol.2	Sol.1
		$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$
Time to solve a problem of size:	10	ε	ε	ε	ε
	50	ε	ε	ε	ε
	100	ε	ε	ε	ε.
	1000	ε	ε	0.02s	4.5s
	10000	ε	0.01s	2.1s	75m
	100000	0.04s	0.12s	3.5m	52d
	1 mil.	0.42s	1.4s	5.8h	142yrs.
	10 mil.	4.2s	16.1s	24.3d	140000yrs.
Max size problem solved in	1s	2.3 mil.	740000	6900	610
	1m	140 mil.	34 mil.	53000	2400
	1d	200 bil.	35 bil.	2 mil.	26000
time if n increases:	x 2	x 2	x 2+	x 4	x 8

Points to Remember

- Even with today's fast processors, designing better algorithms does matter.
- Asymptotic notation is a relevant measure of running time of algorithms.
  - It allows us to easily analyze and compare algorithms.
    - Working at a higher level of abstraction we do away with implementation details and computer-specific issues.
- For a single problem there can be several solutions with different time complexities.
  - Sometimes a better solution can be even easier to implement.
- Polynomial-time algorithms are much better than exponential ones.

Design Suggestions

Bentley has the following suggestions:

1. When possible, try to <u>save state</u> to avoid recomputation.
2. <u>Preprocess data</u> to build useful data structures.
3. Investigate the possibility of using a <u>divide-and-conquer</u> strategy.
4. Consider the use of a <u>scanning</u> algorithm.
   - Problems on arrays can often be solved by asking: “How can I extend a solution for X[1… i-1] to a solution for X[1…i]?”
5. Consider the use of a <u>cumulative vector</u>.
   - In Bentley’s problem we calculate a value as the difference of two sums.
6. Calculation of <u>lower bounds</u> (“Are we home yet?”)
   - You can only be sure that you have the best algorithm if you can prove a lower bound for the problem (i.e. the general solution to this problem **must** take **at least** this many steps).