1 Motivation problem

[PP, Chapter 7]

Bentley's problem:

- Given an array $A[1..n]$ of integer numbers.
- Find contiguous subarray which has the largest sum.

Example:

```
31 -41 59 26 -53 58 97 -93 -23 84
       ^^^^^^^^^^^^^^^
             187
```

Quiz questions:

- What if all numbers are positive?
- What if all numbers are negative?

(Simple) Solution 1: Try all possible subarrays and choose 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;
```

Recall: O notation for measuring how running time grows with the size of the output. (cs240)
Informally: Running time is $O(f(n))$ if it is "proportional" to $f(n)$ for the input of size n.

Time: $O(n^3)$

Q: Can we do better?

Solution 2a: We don't need to recompute sum from scratch every time.

```
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;
```

Time: $O(n^2)$

Solution 2b: We can compute sum in constant time if we do a little bit of pre-computation.

Let $B[i]$ be the sum of $A[1] + \ldots + A[i]$.
Then $A[i] + \ldots + A[j] = B[j] - B[i-1]$.

```
// 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];
```

Time: $O(n^2)$

Solution 3 (Divide-and-conquer):

Recall MergeSort: (cs240)
To sort the array:

- Divide an array into two equally-sized parts
- Sort each part separately
- Solution is obtained by "merging" the smaller solutions

The same approach can 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 must be going "through the midle"; therefore:
  - Find the maximum subarray for left part ($\max_L$) and right part ($\max_R$)
  - Find the maximum subarray going "through the middle" ($\max_M$) — this can be done in linear time $O(n)$
  - $\max\{\max_L, \max_R, \max_M\}$ is the solution.

Examples:

```
           max_M=32+155=182
          vvvvvvvvvvvvvvvvv
31 31 -70 59 26 -53 | 58 97 -90 -90 80 80
          ^^^^^                     ^^^^^
         max_L=85                  max_R=160


           max_M=2+155=157
          vvvvvvvvvvvvvvvvv
31 31 -70 59 26 -83 | 58 97 -90 -90 80 80
          ^^^^^                     ^^^^^
         max_L=85                  max_R=160
```

```
                  max_M=0+155
                    vvvvvvv
31 31 -70 59 26 -93 | 58 97 -90 -90 80 80
          ^^^^^                    ^^^^^
         max_L=85                 max_R=160
```

Time: $O(n \log n)$, as in MergeSort.
(If interested in the details, have a look at PP, chapter 7)

Solution 4:

- $maxsol_i$ be the maximum sum subarray of array $A[1..\boxed{\text{i}}]$.
- $tail_i$ be the maximum sum subarray that ends at position i.

What is the relationship between $maxsol_i$ and $maxsol_{i-1}$?

$$maxsol_i = \max \begin{cases} maxsol_{i-1}, \\ tail_i, \end{cases}$$

$$tail_i = \max \begin{cases} tail_{i-1} + A[i], \\ 0. \end{cases}$$

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

Time: O(n)

Time comparison

- Solutions implemented in C.
- Some of the values are measured (on Pentium II), some of them are estimated from the other measurements.
- Solution 0 is a fictitious exponential-time solution (just for comparison with others)
- ε means under 0.01s

		Sol.4 $O(n)$	**Sol.3** $O(n \log n)$	**Sol.2** $O(n^2)$	**Sol.1** $O(n^3)$	**Sol.0** $O(2^n)$
Time to	10	ε	ε	ε	ε	ε
solve a	50	ε	ε	ε	ε	2 weeks
problem	100	ε	ε	ε	ε	2800 univ.
of size	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	142yr	—
	10 mil.	4.2s	16.1s	24.3d	140000yr	—
Max size	1s	2.3 mil.	740000	6900	610	33
problem	1m	140 mil.	34 mil.	53000	2400	39
solved in	1d	200 bil.	35 bil.	2 mil.	26000	49
Increase in	+1	—	—	—	—	×2
time if n	×2	×2	×2+	×4	×8	—
increases						

Points to take home:

- Even with today's fast processors, designing better algorithms matters.
- Asymptotic notation is a relevant measure of the running time of algorithms. It allows us to easily analyze and compare algorithms and abstract away 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.