1 Motivation problem

[Ben1, 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.
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 $A[1] + \cdots + A[i]$.
Then $A[i] + \cdots + A[j] = B[j] - B[i-1]$.

```
// 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:
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, 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 solve a problem of size ...	10	ε	ε	ε	ε	ε
	50	ε	ε	ε	ε	2 weeks
	100	ε	ε	ε	ε	2800 univ.
	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 problem solved in	1s	2.3 mil.	740000	6900	610	33
	1m	140 mil.	34 mil.	53000	2400	39
	1d	200 bil.	35 bil.	2 mil.	26000	49
Increase in time if n increases	+1	—	—	—	—	×2
	×2	×2	×2+	×4	×8	—

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.

2 Analyzing Running Time of Algorithms

[BB chapters 2,3.1-3.3,4.1-4.4] or [Par sections 1.1-1.4] or [CLRS2 chapters 1-3]

2.1 Problem-Algorithm-Instance-Running Time

We design algorithms to solve **problems**:

- What are valid inputs (or *instances*)?
- What output should we get for each input?

In Bentley's problem:

- *Input: any array of integers*
- *Output: subarray with maximum sum*

An algorithm solves the problem if for every valid instance of the problem it finds a valid output.

Running time:

- Running time of an **algorithm** A **on instance** x is the $\boxed{\text{time}}$ that algorithm A requires to solve input x (denote $T_A(x)$).
- (Worst-case) running time of an **algorithm** A is a function of the size of the input instances, where $T_A(n)$ is the largest time required to solve instance an of $\boxed{\text{size}}$ n, or

$$T_A(n) = \max\{T_A(x) \mid |x| = n\}$$

- Time **complexity of a problem** is a running time of $\boxed{\text{the best}}$ algorithm solving the problem.

Note: We did not yet define $\boxed{\text{boxed terms}}$.

Size of the instance

Formally: number of bits needed to encode the input.

This is often too complicated – we choose some other (more natural) parameter of the input.

In Bentley's problem: sum of number of bits needed to encode all the numbers in the array.

In Bentley's problem: number of elements in the array.

Running time on the instance

To simplify theoretical analysis, we need to abstract away details of the computation (exact speed of the processor, disk, memory, caching, etc.); therefore we count the number of **elementary operations.**

([PP] talks about how to account for some of these issues.)

Elementary operation is an operation whose time can be bounded by a constant that depends **only** on the implementation of the operation (either in hardware on software) and not on the inputs of the operation.

- **Elementary operations:** simple arithmetic operations, comparisons (problems when large numbers or arbitrary precision arithmetic is allowed), program flow control operations, etc.
- **Not elementary operations:** maximum in an array of numbers, does string contain a given substring?, concatenation of two strings, factorial, etc. (beware: many programming languages offer constructs that are not elementary operations)

Note: Notion of elementary operation depends somewhat on the computational model. For most of the course an intuitive notion of the elementary operation will be satisfactory. We will introduce a formal model of computation later.

2.2 Asymptotic Notation

. . . or how to compare algorithms.

Definition 1. *Function $f(n)$ is in $O(g(n))$ iff there exist $c > 0$ and $n_0 > 0$ such that:*

$$(\forall n > n_0)(0 \leq f(n) \leq cg(n))$$

Notation: $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.

The following claims can be proven from the definition (some of them are on the assignment):

- if $f(n) \in O(g(n))$ and $c > 0$ is a constant
  then $cf(n) \in O(g(n))$
- if $f(n) \in O(f'(n))$ and $g(n) \in O(g'(n))$ then
  - $f(n) + g(n) \in O(f'(n) + g'(n))$
  - $f(n)g(n) \in O(f'(n)g'(n))$
- **Maximum rule.** If $t(n) \in O(f(n) + g(n))$
  then $t(n) \in O(\max(f(n), g(n)))$
- **Transitivity.** If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$
  then $f(n) \in O(h(n))$

Examples:

- $3.8n^2 + 2.6n^3 + 10n \log n \in O(n^3)$ (we use as simple form as possible)
- $10^{100}n \in O(n)$
- $(n + 1)! \in O(n!)$ true or <u>false</u>?
- $2^{2n} \in O(2^n)$ true or <u>false</u>?
- $n \in O(n^{10})$ <u>true</u> or false? (*)

Example (*) shows that we need other asymptotic notations.

Notation	Definition	Analogy to arithmetic comparisons
$f(n) \in O(g(n))$	There exists $c > 0$ and $n_0 > 0$ s.t. $(\forall n > n_0)(0 \leq f(n) \leq cg(n))$	$\leq$
$f(n) \in \Omega(g(n))$	There exists $c > 0$ and $n_0 > 0$ s.t. $(\forall n > n_0)(f(n) \geq cg(n) \geq 0)$	$\geq$
$f(n) \in \Theta(g(n))$	$f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$	$=$
$f(n) \in o(g(n))$	For any $c > 0$ there exists $n_0 > 0$ s.t. $(\forall n > n_0)(0 \leq f(n) < cg(n))$	$<$
$f(n) \in \omega(g(n))$	For any $c > 0$ there exists $n_0 > 0$ s.t. $(\forall n > n_0)(f(n) > cg(n) \geq 0)$	$>$

How to prove that $f(n) \notin O(n)$?

Definition:

$$f(n) \in O(g(n)) \Leftrightarrow (\exists c > 0)(\exists n_0 > 0)(\forall n > n_0)(0 \leq f(n) \leq cg(n))$$

Negation:

$$f(n) \notin O(g(n)) \Leftrightarrow (\forall c > 0)(\forall n_0 > 0)(\exists n > n_0)(f(n) < 0 \text{ or } f(n) > cg(n))$$

Example: $(n+1)! \notin O(n!)$

It holds: $(n+1)! = (n+1) \cdot n!$. Now, take any $c > 0$ and $n_0 > 0$ and take $n = \lceil c \rceil \lceil n_0 \rceil$. Then:

$$(\lceil c \rceil \lceil n_0 \rceil + 1)(\lceil c \rceil \lceil n_0 \rceil)! > c(\lceil c \rceil \lceil n_0 \rceil)!$$

Note: The negation is not identical to the definition of $\omega(g(n))$.

- if $f(n) \in \omega(g(n))$ then $f(n) \notin O(g(n))$
- if $f(n) \in O(g(n))$ then $f(n) \notin \omega(g(n))$
- but there are functions where $f(n) \notin O(g(n))$ and $f(n) \notin \omega(g(n))$