Sum of a Linear Array

- Given an array

$$\mathbf{a} = (\, a_i : i = 0, 1, \ldots, n-1 \,),$$

compute the sum of its elements:

$$s = a_0 + a_1 + \ldots + a_{n-1}$$

- Elementary operations:
  - fetching a_i from memory: n operations per array
  - addition: $n-1$ operations per array

Sum of a Linear Array

- Linear computing time: for $n >> 1$

$$\mathrm{T}(n) = c \cdot n$$

- Factor c depends on the computer, operating system, programming language, compiler, *etc*
- Linear relative changes:

$$\mathrm{T}(\alpha \cdot n) = \alpha \cdot \mathrm{T}(n)$$

Sums of Contiguous Subsequences

- Given an array

$$(a_i : i = 0,1, \ldots, n-1)$$

of size $n = 2m$, compute $m + 1$ sums:

$$s_k = a_k + a_{k+1} + \ldots + a_{k+m-1};\ k = 0, \ldots, m$$

of all contiguous subsequences of size m

- **Brute force computation**: $c \cdot m$ operations per subsequence; in total: $c \cdot m \cdot (m+1)$ operations

Sums of Contiguous Subsequences

- **Brute-force program** (*two nested loops*):

```
...
int[ ] s = new int[ m + 1 ];
for( int k = 0; k <= m; k++ ) {     // m + 1 sums
  s[ k ] = 0;
  for( int i = 0; i < m; i++ )      // each sum: c · m
    s[ k ] += a[ k + i ];           //      operations
}
```


Brute-force Quadratic Approach

- Quadratic time (*two nested loops*):

$$\mathrm{T}(n) = c \cdot \frac{n}{2} \cdot \left(\frac{n}{2} + 1 \right) \cong c' \cdot n^2$$

- Quadratic relative changes:

$$\mathrm{T}(n) = n^2 \cdot \mathrm{T}(1)$$

Brute-force Quadratic Approach

- Quadratic time complexity is due to reiterated computations in the innermost loop

$$s_k = a_k + \underline{a_{k+1} + \ldots + a_{k+m-1}}$$

$$s_{k+1} = \underline{a_{k+1} + \ldots + a_{k+m-1}} + a_{k+m}$$

Getting Linear Computing Time

- Exclude reiterated computations of the "brute-force" algorithm:

$$s_{k+1} = s_k + a_{k+m} - a_k$$

- As a result, the linear computing time:

$$T(n) = c \cdot (m + 2m) = 1.5c \cdot n$$

Getting Linear Computing Time

- Better program (two successive loops):

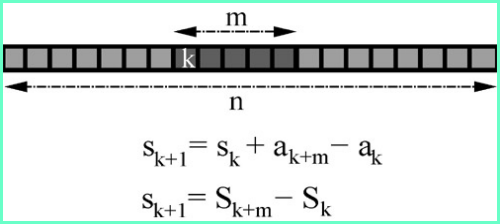
```
...
int[ ] s = new int[ m + 1 ];
s[ 0 ] = 0;
for( int i = 0; i < m; i++ )   // first sum: m operations
  s[ 0 ] += a[ i ];
for( int k = 1; k <= m; k++ )  // 2 operations per sum
  s[ k ] = s[ k – 1 ] + a[ k + m – 1] - a[ k – 1 ];
```


Computing Time for T(1)=1μs

Array size	n	2,000	2,000,000
Size / number of subsequences	m / $m+1$	1,000 / 1,001	1,000,000 / 1,000,001
Brute-force (*quadratic*) algorithm	$T(n)$	2 s	> 23 days
Efficient (*linear*) algorithm	$T(n)$	1.5 ms	1.5 s

Practical Usage: Data Averaging

Data smoothing in a moving 1D window:



Data Averaging

Window size: $m \in \{ 2, 3, \ldots, {}^{n}/_{2} \}$

Partial sums: $S_k = a_0 + \ldots + a_k$

Brute-force approach: $T(n) = c \cdot m \cdot n \cong C \cdot n^2$
if m is a fraction of n

Brute-force computing time might be even cubic: $T(n) = C \cdot n^3$ when the averaged data should be compared for many windows $2 \le m \le {}^{n}/_{2}$

Faster Data Averaging

- Sequential modification of the current sum:
  $s_{k+1} = s_k + a_{k+m} - a_k$
  $T(n) = m + 2 \cdot (n - m)$ per window ($m \le {}^{n}/_{2}$)
- An extra memory for storing n partial sums
  $S_k = s_0 + s_1 + \ldots + s_k;\ k = 0, 1, \ldots, n$
  allows to find the sum of each subsequence with only 1 subtraction per window: $s_{k+1} = S_{k+m} - S_k$

Estimating Running Time

- Simplifying assumptions:
  - all elementary statements / expressions take the same amount of time to execute
    - e.g., simple arithmetic assignments
    - `return`
- Loops increase in time linearly as

$$k \cdot T_{\text{body of a loop}}$$

where k is number of times the loop is executed

Estimating Running Time

- Conditional / switch statements are more complicated (because it is necessary to account for frequencies of branches)
- Function calls:

$$T_{\text{function}} = \Sigma\, T_{\text{statements in function}}$$

- Function composition:

$$T(f(g(n))) = T(g(n)) + T(f(n))$$

Estimating Running Time

- Function calls: $T = \Sigma\, T_{\text{statement } i}$

... x.myMethod(y, ...);
public void myMethod(int a, ...){
statements 1, 2, … , N }

- Function composition $T(f(g(n)))$
  - Computation of $x = g(n) \rightarrow T(g(n))$
  - Computation of $y = f(x) \rightarrow T(f(n))$

Time to Sum Contiguous Sequences

- Ignore data initialisation code
- "Brute-force" nested loops:

$$T(n) = m \cdot (m+1)$$
$$= {}^{n}/_{2} \cdot ({}^{n}/_{2} + 1)$$
$$= 0.25\, n^2 + 0.5\, n$$

- For a large n, $T(n) \cong 0.25\, n^2$
  - e.g., if $n > 500$, $0.5\, n < 0.4\%$ of $T(n)$

Time to Sum Contiguous Sequences

- Factor $c = 0.25$ is referred to as a "constant of proportionality"
- This factor does not effect the behaviour of the algorithm for a large n:
  - 10% increase in n leads to a 20% increase in $T(n)$

Quadratic vs linear time

$T(n) = 0.25n^2 + 0.5n$				
n	$T(n)$	$0.25n^2$	$0.5n$	
10	30	25	5	16.7%
50	650	625	25	3.8%
100	2550	2500	50	2.0%
500	62750	62500	250	0.4%
1000	250500	250000	500	0.2%