Mathematical Concepts and Performance Measures

The main point of this chapter is to develop a system for evaluating and comparing ***algorithms***, not programs. For this reason, we do not measure performance differences between two implementations of the same algorithm, nor are we concerned with precise calculations of running times, since these depend on the implementation, the compiler, the operating system, and the machine architecture. Instead, we devise a means of evaluating the actual algorithm, independent of its implementation. We must work at a higher level of abstraction, throwing away details of implementation that do not affect how "fast" it is. We have yet to define what "fast" means.

1 Mathematical Background

First, we define a method of measuring how "fast" functions of a single variable can grow. Use your intuition here. Consider the three non-decreasing functions

$f(x) = x^2$

$g(x) = x^3$

$h(x) = 5x^2$

Your intuition should tell you that as x gets larger and larger, the function $g(x)$ grows faster and faster than the others. Furthermore, for any value of x, $h(x)$ will always be exactly $5f(x)$. So whatever the rate of growth of $f(x)$, $h(x)$ is growing at the same rate as $f(x)$. They stay in a kind of lock step, with h and f proportionally the same as x marches towards infinity. On the other hand, as x increases, clearly $g(x)$ gets larger and larger than both $f(x)$ and $h(x)$. To see this, look at the first six integer cubes: 1, 8, 27, 64, 125, 216; whereas the first six integer squares are 1, 4, 9, 16, 25, 36. Even the first six values of $h(x)$ are overtaken by the faster growing $g(x)$: 5, 20, 45, 80, 125, 180.

The point is that whatever means we use to measure the relative rates of growth of functions, it ought to ignore constant factors such as the 5 above, and must rank functions like cubics as being faster of quadratics. The following definitions do just that.

2 Definitions of Asymptotic Rates of Growth

The word ***asymptotic*** is an adjective that means, "approaching a limit." In computer science, the limit is usually infinity, and the adjective is applied to the behavior of a function. ***Asymptotic analysis*** refers to the study of the limiting behavior of algorithms as their inputs approach infinity.

The three operators, "big O", "big omega Ω", and "theta Θ", define sets of functions. In other words, $O(f(n))$ is a set of functions (of one variable) that are related to f in some precise way, $\Omega(f(n))$ is a different set of function related to f. It is good to think of "big O", "big omega Ω", and "theta Θ" as operators that define sets of functions. In fact we will use the membership symbol $\in$ to indicate this.

The formal definitions of these operators are as follows.

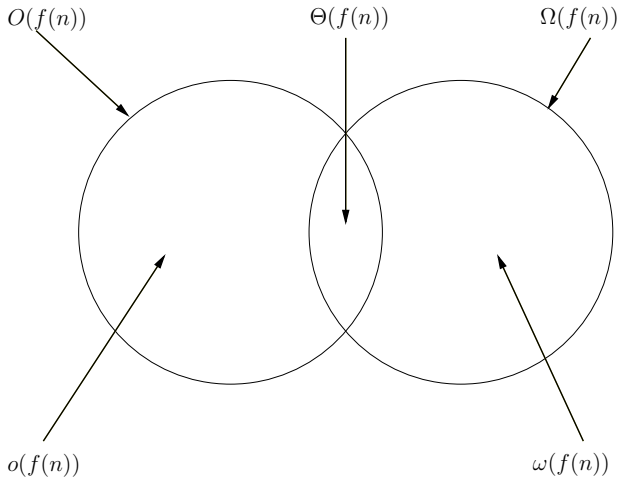

Figure 1: Relationships among the different classes of rates of growth.

Big O

$t(n) \in O(f(n))$ if there exist two numbers m and c such that $m \geq 0$ and $c \geq 0$ and $t(n) \leq cf(n)$ for all $n \geq m$.

In other words, $O(f(n))$ is the set of all functions $t(n)$ such that there is a constant c such that $t(n) \leq cf(n)$ for sufficiently large values of n. You could say that $O(f(n))$ is the set of functions that grow no faster than f.

Big Omega Ω

$t(n) \in \Omega(f(n))$ if there exist two numbers m and c such that $m \geq 0$ and $c > 0$ and $t(n) \geq cf(n)$ for all $n \geq m$.

In other words, Ω defines something like the opposite of big-O. The functions $t(n)$ in $\Omega(f(n))$ have the property that for each such $t(n)$ there is a c such that for sufficiently large n, $t(n) \geq cf(n)$. You could say that $\Omega(f(n))$ is the set of functions that grow no slower than f.

Theta Θ

$t(n) \in \Theta(f(n)$ iff $t(n) \in O(f(n))$ and $t(n) \in \Omega(f(n))$.

The set $\Theta(f(n))$ is actually the intersection of the first two sets. It consists of those functions that grow no faster than f and grow no slower than f. You could say that it is the set of functions that grow at the same rate as f. Remember to be careful with this. The functions $1000f(n)$ and $f(n)/10000$ are each in $\Theta(f(n))$.

Little o

$t(n) \in o(f(n)$ iff $t(n) \in O(f(n))$ and $t(n) \notin \Theta(f(n))$.

The set $o(f(n))$ is the intersection of $O(f(n))$ and the complement of $\Theta(f(n)$. It consists of functions that grow strictly slower than f. For example, $n \in o(n^2)$ but $2n^2 \notin o(n^2)$.

Little Omega ω

$t(n) \in \omega(f(n))$ iff $t(n) \in \Omega(f(n))$ and $t(n) \notin \Theta(f(n)$.

The set $\omega(f(n))$ is the intersection of $\Omega(f(n))$ and the complement of $\Theta(f(n)$. It consists of functions that grow strictly faster than f. For example, $n^3 \in \omega(n^2)$ but $2n^2 \notin \omega(n^2)$.

See Figure 1 for a visual depiction of the relationships among these different functions.

2.1 Some Implications Of The Definitions

Lemma 1. *If $t(n) \in O(f(n))$ and $s(n) \in O(g(n))$ then*

1. $s(n) + t(n) \in O(f(n) + g(n))$
2. $s(n) \cdot t(n) \in O(f(n) \cdot g(n))$

Proof. Because $t(n) \in O(f(n))$, there is an integer m and a constant c such that for all $n > m$, $t(n) \leq cf(n)$. Similarly, there are constants k and d such that $s(n) \leq gd(n)$ for all $n > k$. Let $C = \max(c, d)$ and let $M = \max(m, k)$. Then $s(n) + t(n) \leq Cf(n) + Cg(n) = C(f(n) + g(n))$ for all $n > M$, proving that $s(n) + t(n) \in O(f(n) + g(n))$.

Similarly, $s(n) \cdot t(n) \leq Cf(n) \cdot Cg(n) = C^2 f(n) \cdot g(n)$ for all $n > M$. In this case we use C^2 as the constant, and this shows that $s(n) \cdot t(n) \in O(f(n) \cdot g(n))$. □

Define

$$\max(f(n), g(n)) = \begin{cases} f(n) & if\ g(n) \in O(f(n))\ and\ f(n) \notin O(g(n)) \\ g(n) & if\ f(n) \in O(g(n)) \\ undefined & otherwise \end{cases}$$

In other words, $\max(f(n), g(n))$ is the faster growing function of $f(n)$ and $g(n)$. It is not hard to show that, if max is defined for f and g then $O(f(n) + g(n)) = O(\max(f(n), g(n)))$. The reason that it may not be defined is that one or the other of $f(n)$ and $g(n)$ may be ***oscillating***. See below for an example.

Calculus and the theory of limits can be used to determine the relative growth rates of functions. If necessary, ***L'Hopital's rule*** can be used for solving the limit:

if $\lim_{n\to\infty}(f(n)/g(n))=$	then all of these are true statements:
0	$f(n) \in O(g(n))$ and $f(n) \in o(g(n))$
$c \neq 0$	$f(n) \in \Theta(g(n))$
∞	$f(n) \in \Omega(g(n))$ and $f(n) \in \omega(g(n))$
oscillating	there is no limit and there is no relationship.

For an example of an oscillating fraction, let

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

and let $g(n) = n$. Then the fraction $f(n)/g(n)$ alternates between 1 and $1/n$ as n approaches infinity, so there is no limit.

2.2 Some Growth Rate Relationships

In the following, the symbol < means that the function to the left is little-o of the function to the right.

$$c < \log N < \log^k N < N < N \log N < N^2 < N^3 < N^{3+k} < 2^N < 3^N < \ldots < N!$$

where the base of the log does not matter, and k is any integer greater than 1.

3 Model of Computation

When analyzing running time, we assume that the algorithm runs on some theoretical, abstract computer. It is usually a computer based on the ***Von Neumann architecture***. The Von Neumann architecture, named after John von Neumann[1], the mathematician who invented it, is one in which both the program and its data are stored in memory, and instructions are executed one after the other in sequence, fetching operands from memory. This is exactly the kind of computer you use today. Before Von Neumann, computers were single-purpose machines whose programs were hardware controlled. Other abstract models yield different running times.

Further assumptions about program execution are that

- all instructions take exactly the same amount of time
- memory is infinite
- all instructions are "simple" instructions that act on scalars, not vectors.

These assumptions are designed to simplify the analysis process without any loss of correctness. While it is true that some instructions take longer than others, for example, the difference is a constant factor that gets ignored anyway in the rate of growth analysis.

3.1 Input Size

Every input is assumed to have a positive integer size that depends on the particular problem to be solved. For example, the sorting problem sorts lists of things. The number of elements in the list is the size of the input, not the lengths of the items to be sorted, or their combined lengths. If the problem is searching for the occurrence of one string in a second string, the number of characters in the first string and the number of characters in the second string are the two input sizes. Different algorithms may depend on their sum, or product, or some other function of the two sizes. If the problem is to search for a keyword in a dictionary, the input size is not the length of the keyword, but the number of words in the dictionary.

[1] John von Neumann, "First Draft of a Report on the EDVAC", United States Army Ordinance Department and the University of Pennsylvania, 1945.

3.2 What to Analyze

The running time of an algorithm depends on the input it is given. For some inputs it might be fast and for others, it might be slow. We are usually interested in knowing the worst it can possibly do, because then we can plan conservatively. Sometimes though, we want to know the *average* running time. The idea of *average* is really useless because it treats all inputs as equally likely to occur, which is never true. What is more useful is the ***expected*** running time, which is the weighted average, i.e., each input is weighted by its probability of occurring. This is also a little silly since we rarely are able to weight the inputs realistically. Therefore, average running time is used, but everyone knows it is only an approximation. I will use the following notation the running times for an input of size N:

$T_{avg}(N)$ average (not expected value)

$T_{expected}(N)$ probabilistic analysis taking into account distribution of inputs

$T_{worst}(N)$ worst case

3.3 Running Time Calculations

These are rules for analyzing the running time of programs as representations of algorithms – the rules are designed to ignore details of implementation.

Remember that we usually ignore constant multiples and constant terms.

3.3.1 For Loops

The running time of a ***for loop*** is at most the running time of the statements inside the loop multiplied by the number of iterations of the loop.

3.3.2 Nested Loops

From the preceding statement, it follows that the running time of a statement inside ***nested loops*** is the running time of the statement multiplied by the product of the number of iterations of the loops. For example, the triply nested pseudo-code loop

```
for i = 1 to n
    for j = 1 to 2n
        for k = 1 to 3n
            S
```

runs in time proportional to the product of the running time of S and n^3. If the running time of S is $r(S)$,then in order notation it is $O(r(S) \cdot n^3)$.

3.3.3 Consecutive Statements

The total running time of a sequence of statements is the sum of the running times of the statements. For example, the sequence

```
S1 ; S2 ; S3 ; ... ; Sn
```

has a running time equal to the sum of the running times of `S1`, `S2`, `S3`, ..., `Sn`. If order notation is being used to represent running time, then it is usually just the maximum of the running times of the individual statements that is reported.

3.3.4 If/else

The running time of an ***if/else*** statement

```
if (condition)
    S1
else
    S2
```

is at most the running time of the evaluation of the condition plus the sum of the running times of `S1` and `S2`.

If we are interested in worst case analysis, then we have to assume that both branches of the statement are reachable unless we can prove that one is never reached. Under this assumption, the running time of the ***if/else*** statement is the maximum of the condition, `S1`, and `S2`.

3.3.5 Function Calls

If a sequence of statements contains function calls, the running time of the calls must be determined first. For recursive functions, the analysis can get difficult, requiring a recurrence relation to be solved.

Example

```
long fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Let $T(n)$ be the running time of this function given input n. Then when the argument to the function is 0 or 1, the function returns without a recursive call, so the running time is a constant, which we choose to be 1:

$$T(0) = T(1) = 1$$

If the input is greater than 1, then two recursive calls are made and some constant work is done as well.

$$T(n) = T(n-1) + T(n-2) + 1$$

The constant 1 is for the evaluation of the condition and the execution of the `return` statement. It indicates that besides the time for each recursive call, there is ***some*** work that must be done in this call to the function, no matter how small. This is a recurrence relation.

To solve this recurrence we make a few observations:

1. $T(n) \geq \text{Fib}(n)$ because it adds 1 each time.
2. $\text{Fib}(n) \geq (3/2)^n$ for all $n > 4$, which we now prove.

Proof. We prove this by induction on n.

Base Case: $\text{Fib}(5) = 8 > (3/2)^5 = 7.59375$ and $\text{Fib}(6) = 13 > (3/2)^6 = 11.390625$ proving it is true when $n = 5$ and when $n = 6$.

Inductive Hypothesis: Assume that for any $n > 6$, if $k < n$ then $\text{Fib}(k) \geq (3/2)^k$. Then we have

$$\begin{aligned}
\text{Fib(n+1)} &= \text{Fib (n)+Fib (n-1)} \\
&> (3/2)^n + (3/2)^{n-1} \\
&= (3/2)^{n-1}((3/2)+1) \\
&= (3/2)^{n-1} \cdot 5/2 \\
&> (3/2)^{n-1} \cdot 2.25 \\
&= (3/2)^{n-1} \cdot (3/2)^2 \\
&= (3/2)^{n+1}
\end{aligned}$$

which proves it is true for $n + 1$. □

This shows that $T(n)$ grows exponentially. It doesn't matter how bad it is since that is bad enough. Recursion is not a very good way to compute Fibonacci numbers.

4 Example: The Maximum Subsequence Problem

This is a very nice problem to study for several reasons. One is that it shows how some cleverness can be used to replace a poorly performing solution by an extremely efficient one. Second, it gives us a chance to analyze three very different types of algorithms. Third, it illustrates an important concept in algorithm design, that whenever an algorithm computes a piece of information, it should try to reuse that information as much as possible. You shall see what this means soon.

Roughly stated, the maximal subsequence problem asks you to find a subsequence of a sequence of positive and negative numbers whose sum is the largest among all possible subsequences of the sequence. For example, given this sequence of numbers:

```
1, 2, -4, 1, 5, -10, 4, 1
```

you can verify that the subsequence 1, 5 has a sum of 6, and no other subsequence has a sum greater than 6.

Formally, given a sequence of possibly negative integers A_1, A_2, ..., A_N , find

$$\max_{i \le j} \left\{ 0, \sum_{k=i}^{j} A_k \right\}$$

For those unfamiliar with the notation, it means, find the maximum sum of each possible sequence of numbers from A_i to A_j for all pairs of indices i and j such that $i \le j$, and choose zero if all are negative.

Example. The sequence -2, 11, -4, 13, -5, -2 has a maximal subsequence whose sum is 20. The following table proves this. There is a row for each possible starting value, and a column for each possible length of sequence that starts at that value. The largest sum is 20, for the sequence of length 3 starting at 11.

	Length of Sequence					
Starting value of sequence	**1**	**2**	**3**	**4**	**5**	**6**
-2	-2	9	5	18	13	11
11	11	7	20	15	13	0
-4	-4	9	4	2	0	0
13	13	8	6	0	0	0
-5	-5	-7	0	0	0	0
-2	-2	0	0	0	0	0

The brute force method that is described later was used to make the table above.

4.1 Rules for Evaluating Running Time

1. Never count the time to read input, since this is always $O(n)$ for inputs of size n and it will hide the real running time of efficient algorithms.
2. Ignore constant multiples – use only order notation for rate of growth problems.

4.2 Solutions to Maximal Subsequence Problem

4.2.1 Brute Force

The following is a C++ function that solves this problem using brute force. It just checks all possible starting positions and all sequence lengths at that position, and adds the numbers in the subsequence:

```
int maxSubSeqSum( const  vector<int> & a)
{
    int max = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i; j < a.size(); j++) {
            int sum = 0;
            for (int k =i ; k <= j; k++)
                sum += a[k];
            if ( max < sum)
            max = sum;
        }
    return max;
}
```

4.2.2 Analysis

The innermost loop executes $(j-i+1)$ times. This ranges over all j from i to N. Thus, the body of the loop is executed

$$\sum_{j=i}^{N-1}(j-i+1) = \sum_{j=0}^{N-1-i}(j+1) = \sum_{j=1}^{N-i} j = \frac{(N-i+1)(N-i)}{2}$$

times. The value of i ranges from 0 to $N-1$. Thus, the total number of executions is

$$\begin{aligned}
\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2} &= \sum_{i=1}^{N}\frac{(N-(i-1)+1)(N-(i-1))}{2} \\
&= \sum_{i=1}^{N}\frac{(N-i+2)(N-i+1)}{2} \\
&= \sum_{i=1}^{N}\frac{N^2+3N+2-(2Ni+3i)+i^2}{2} \\
&= \frac{1}{2}\left(\sum_{i=1}^{N} N^2+3N+2\right) - \frac{2N+3}{2}\left(\sum_{i=1}^{N} i\right) + \frac{1}{2}\left(\sum_{i=1}^{N} i^2\right) \\
&= \frac{N(N^2+3N+2)}{2} - \frac{2N+3}{2}\left(\frac{N(N+1)}{2}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= \frac{6N(N^2+3N+2)}{12} - \left(\frac{3N(N+1)(2N+3)}{12}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= \frac{6N(N+1)(N+2)}{12} - \left(\frac{3N(N+1)(2N+3)}{12}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= N(N+1)\left(\frac{6(N+2)-3(2N+3)+(2N+1)}{12}\right) \\
&= N(N+1)\left(\frac{2N+4}{12}\right) \\
&= \frac{N(N+1)(N+2)}{6} = \frac{N^3+2N^2+2N}{6}
\end{aligned}$$

which is $O(N^3)$. Why didn't I just use the nested loop rule? Because it was not clear that the loops were each $O(n)$, was it?

4.2.3 A Divide-and-Conquer, Recursive Solution

A more efficient solution can be obtained from the following observation:

A maximal subsequence is either entirely within the first half, or entirely within the second half, or it straddles the two halves. If it straddles the two halves, it can be broken into two pieces:

1. the sequence with the largest sum entirely within the first half that contains the last element of the first half.
2. the sequence with the largest sum entirely within the last half that contains the first element of the last half.

Thus, we can find the sequences in each half that satisfy these conditions, add them up and compare to the sequences found recursively in each half. We just take the max of all of them.

```
/**
Recursive maximum contiguous subsequence sum algorithm. Finds maximum sum
    in subarray spanning a[left..right]. Does not attempt to maintain
    actual best sequence.
*/
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base cases
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center      = ( left + right ) / 2;
    /* Recursion here */
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0;
    int leftBorderSum    = 0;

    /* find the sum of every sequence ending at a[center]
       and starting at i, where i = center, center-1, center-2,...
       and save the maximum sum in MaxLeftBorderSum */
    for ( int i = center; i >= left; i-- ) {
        leftBorderSum += a[ i ];
        if ( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    /* Do the analogous thing to the right-hand side of the center */
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for ( int j = center + 1; j <= right; j++ ) {
        rightBorderSum += a[ j ];
```

```
        if ( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }
        return max3( maxLeftSum, maxRightSum,
              maxLeftBorderSum + maxRightBorderSum );
}

/* Driver for divide-and-conquer maximum contiguous subsequence sum
   algorithm. */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

Analysis The algorithm makes two recursive calls with half size arrays. It also processes each element once in a pair of loops. The recurrence is

$T(1) = 1$

$T(N) = 2T(N/2) + cN$

If we "telescope" this recurrence relation, we get

$$\begin{aligned} T(N) &= 2(2T(N/4) + cN/2) + cN \\ &= 4T(N/4) + cN + cN \\ &= 4(2T(N/8) + cN/4) + 2cN \\ &= 8T(N/8) + 3cN \\ &= \ldots \\ &= 2^k T(N/2^k) + kcN \end{aligned} \tag{1}$$

The telescoping stops when $N/2^k = 1$ which occurs when $N = 2^k$ or when $k = \log_2 N$. Substituting $\log_2 N$ for k in Eq. 1, we get

$$\begin{aligned} T(N) &= N \cdot T(1) + \log_2 N \cdot cN \\ &= N + cN \cdot \log_2 N \\ &\in O(N + N \log N) \end{aligned}$$

This solution has a running time that is $O(N \log N)$ which beats $O(N^3)$ significantly! Can we do even better?

4.2.4 Linear Time Solution

A linear time solution to this problem is not hard to find. The previous solutions did not use information they discovered while examining the sequence. This is what I was talking about earlier. As the algorithm scans the string, it can learn so much more than it was doing, and avoid having to recompute or even re-examine previous partial sums.

Concept We take the liberty of letting the notation $a_i \ldots a_{j-1}$ mean both the sequence and its sum, when the meaning is clear. The following assertions form the basis for the algorithm.

1. If a_i is negative, it cannot be the start of a maximal subsequence. (The sequence starting at a_{i+1} would be greater than the one starting at a_i if a_i is a negative number.)

2. More generally than that, any negative subsequence cannot be the start of a maximal subsequence (by the same reasoning as in step 1.)

3. If $a_i \ldots a_{j-1}$ is positive but $a_i \ldots a_j$ is negative, then $a_i \ldots a_j$ cannot be the start of a maximal subsequence. Of course, by step 2, if $a_i \ldots a_j$ is negative, then $a_i \ldots a_j$ cannot be the start of a maximal subsequence, so what does this statement add to that? The next step is the key.

4. Suppose that j is the smallest index greater than i such that $a_i \ldots a_{j-1} \geq 0$ but $a_i \ldots a_j < 0$. In other words, for each index k, $i \leq k < j$, $a_i ... a_k \geq 0$. Consider any p such that $i < p < j$. The sum of the sequence $a_i \ldots a_{j-1}$ can be written as the sum of the numbers from a_i to a_{p-1} and the sum of the numbers from a_p through a_{j-1}:

$$a_i \ldots a_{j-1} = a_i ... a_{p-1} + a_p \ldots a_{j-1} \tag{2}$$

   Since we said that for any k, $i \leq k < j$, $a_i ... a_k \geq 0$, it is true for $k = p - 1$, so $a_i \ldots a_{p-1} \geq 0$. Eq. 2 is of the form

$$a_i \ldots a_{j-1} = X + a_p \ldots a_{j-1}$$

   where X a non-negative number, so it follows that if we subtract it from the right-hand side, the right-hand side stays the same or gets smaller:

$$a_i \ldots a_{j-1} \geq a_p \ldots a_{j-1}$$

   In other words, $a_i \ldots a_{j-1}$ is greater than any of its suffix subsequences. In particular, when we append a_j to both sides of the inequality, it still holds:

$$a_i \ldots a_j \geq a_p \ldots a_j$$

   and since the left hand side is negative, so is the right hand side.

5. This implies that if we have some initial subsequence $a_i \ldots a_{j-1}$ that is positive or zero, and we encounter an a_j that makes the sum negative, we can advance the start index i to the next position after j, i.e., $j + 1$ and start looking for a new maximum subsequence there, because $a_i \ldots a_j$ cannot be the start of a maximum subsequence.

This leads to the following very simple algorithm.

```
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;
    for ( int j = 0; j < a.size( ); j++ ) {
        thisSum += a[ j ];
        if ( thisSum > maxSum )
            maxSum = thisSum;
        else if ( thisSum < 0 )
            // this a[j] made the initial sequence negative - start over
            thisSum = 0;
    }
    return maxSum;
}
```

Analysis It is clearly a linear algorithm since it is a single loop that iterates over every element of the sequence.

You may be tempted to ask whether there exists an even faster solution. If there is not, then we would say that this solution is ***optimal***, meaning the best it can be, because it runs as fast as is possible. But if there does exist a faster solution, this one would not be considered optimal. Can there be a solution that runs in $o(N)$, meaning in less than linear time? Intuitively it is impossible, because any solution must look at each number at least once, and so any solution must run in at least $\Theta(N)$ time.