Introduction to Analysis of Algorithms — **Lecture 1**
CS 4820, Summer 2012 — **July 2, 2012**

Consider the following problem.

Problem 1. *Given a sequence of integers $a_1, \ldots, a_n$, what is the maximum sum of a contiguous subsequence?*

Example 1. *For input*

$$0 \quad 100 \quad -50 \quad -60 \quad 30 \quad -10 \quad 40 \quad 5 \quad 45 \quad -2 \quad 1,$$

the solution is $110 = 30 + (-10) + 40 + 5 + 45$.

We define the empty set to be a valid subsequence: when the entire sequence consists of negative integers only, the correct answer is 0. Problem 1, therefore, can be rewritten as follows:

Problem 2. *Given a sequence of integers $a_1, \ldots, a_n$, compute* $\max\left[\{\sum_{k=i}^{j} a_k \mid 1 \le i \le j \le n\} \cup \{0\}\right]$.

An easy straightforward algorithm is as follows:

Algorithm 1 A straightforward algorithm

```
sol ← 0
for each $i, j$ such that $1 \le i \le j \le n$ do
    if $\sum_{k=i}^{j} a_k$ > sol then
        sol ← $\sum_{k=i}^{j} a_k$
    end if
end for
return sol
```

What is the running time of this algorithm? The given description omits some implementation details of the algorithm, but one possible implementation is shown in Algorithm 2.

Algorithm 2 First attempt

```
sol ← 0
for $i \leftarrow 1, \ldots, n$ do
    for $j \leftarrow i, \ldots, n$ do
        sum ← 0
        for $k \leftarrow i, \ldots, j$ do
            sum ← sum + $a_k$
        end for
        if sum > sol then
            sol ← sum
        end if
    end for
end for
return sol
```

Theorem 1. *Algorithm 2 runs in $O(n^3)$ time.*

Proof. The number of basic operations performed by the algorithm is proportional to

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n}(j-i+1) &= \sum_{i=1}^{n}\sum_{j'=1}^{n-i+1} j' \\
&= \sum_{i'=1}^{n}\sum_{j'=1}^{i'} j' \\
&= \frac{n(n+1)(n+2)}{6} \\
&= O(n^3),
\end{aligned}$$

where the second-to-last line can be proven by induction on n. $\square$

Can we do better?

Suppose $n = 10$. In the iteration where $i = 2$ and $j = 5$, lines 4-7 compute $a_2 + a_3 + a_4 + a_5$; in the next iteration, $i = 2$, $j = 6$, and $a_2 + a_3 + a_4 + a_5 + a_6$ is computed. Note that the partial sum $a_2 + a_3 + a_4 + a_5$ is repeatedly computed in these two iterations. One of the useful questions to ask in order to design an efficient algorithm is whether there are any computations that are repeated by the algorithm, and if so, whether there is a systematic way to reduce such repetitions. Algorithm 3 shows that we can do better.

Algorithm 3 Second attempt

```
 1: $\mathsf{sol} \leftarrow 0$
 2: for $i \leftarrow 1, \ldots, n$ do
 3:     $\mathsf{sum}_{i,i-1} \leftarrow 0$
 4:     for $j \leftarrow i, \ldots, n$ do
 5:         $\mathsf{sum}_{i,j} \leftarrow \mathsf{sum}_{i,j-1} + a_j$
 6:         if $\mathsf{sum}_{i,j} > \mathsf{sol}$ then
 7:             $\mathsf{sol} \leftarrow \mathsf{sum}_{i,j}$
 8:         end if
 9:     end for
10: end for
11: return $\mathsf{sol}$
```

Theorem 2. *Algorithm 3 runs in $O(n^2)$ time.*

Proof omitted.

Theorem 3. *Algorithm 3 is correct.*

Proof. The termination follows from Theorem 2.

The algorithm returns $\max\left[\{\mathsf{sum}_{i,j} \mid 1 \le i \le j \le n\} \cup \{0\}\right]$; hence, it suffices to prove that $\mathsf{sum}_{i,j} = \sum_{k=i}^{j} a_k$.

Claim 1. $\mathsf{sum}_{i,j} = \sum_{k=i}^{j} a_k$.

Proof. By induction on $j - i$.

Base case. If $j - i = 0$, then $\mathsf{sum}_{i,j} = \mathsf{sum}_{i,i-1} + a_i = \sum_{k=i}^{i} a_k$.

Inductive step. Suppose the claim holds for $j - i = d_0$. When $j - i = d_0 + 1$, $\mathsf{sum}_{i,j} = \mathsf{sum}_{i,j-1} + a_j = \left(\sum_{k=i}^{j-1} a_k\right) + a_j = \sum_{k=i}^{j} a_k$, where the second-to-last equality follows from the induction hypothesis. □

□

Can we do better?

The given problem can be interpreted as finding the maximum of $\binom{n+1}{2} = O(n^2)$ values, and in general, in order to determine the maximum among M values, we need to at least look at each of those M values. Therefore, it may appear at a glance that a subquadratic algorithm would be impossible. However, this argument is ignoring the fact that these $\binom{n+1}{2}$ values are not independent from each other; in fact, they are derived from only $O(n)$ integers.

Another useful tool in the design of algorithms is the recursion. In particular, let us consider if an approach similar to that of the Mergesort can be used. We divide the sequence into the left half and the right half, and solve the problem for each subsequence: in the Mergesort, this step corresponds to sorting each half; in the given problem, it corresponds to finding the maximum sum of any contiguous subsequence from each half. Now we need to find the solution for the larger problem, based on the solutions from each half: in the Mergesort, this is done by merging two sorted sublists into a single sorted list. What do we need to do in the given problem? Is Algorithm 4 correct?

Algorithm 4 Third attempt (incorrect)

```
1: function MaxSumSubseq($s$,$t$)
2:     if $s = t$ then
3:         return $\max(a_s, 0)$
4:     end if
5:     $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$
6:     $\mathsf{sol}_l \leftarrow$ MaxSumSubseq($1$,$m$)
7:     $\mathsf{sol}_r \leftarrow$ MaxSumSubseq($m+1$,$n$)
8:     return $\max(\mathsf{sol}_l, \mathsf{sol}_r)$
9: end function
10: return MaxSumSubseq($1$,$n$)
```

If the maximum sum of a contiguous subsequence of $a_s, \ldots, a_t$ is obtained by a subsequence contained in the left half, its sum would be equal to sol_l; if in the right half, sol_r. What if neither is true? Then we know that both a_m and a_{m+1} has to be contained in the subsequence that yields the maximum sum, and this is the only remaining possibility. Therefore, in the 'merging step', we need to find the maximum sum of a contiguous subsequence that contains the boundary between a_m and a_{m+1}. It would take $O((t-s+1)^2)$ time to find such a subsequence by the naïve enumeration, but observe that this maximum sum is given as the sum of the maximum sum of a contiguous subsequence that is on the left half and contains a_m and the maximum sum of a contiguous subsequence that is on the right half and contains a_{m+1}. This observation, combined with the time-saving technique used in Algorithm 3, leads to the $O(n \log n)$-time algorithm shown in Algorithm 5.

Algorithm 5 Third attempt

```
function MaxSumSubseq($s$,$t$)
    if $s = t$ then
        return $\max(a_s, 0)$
    end if
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$
    $\mathsf{sol}_l \leftarrow$ MaxSumSubseq($1$,$m$)
    $\mathsf{sol}_r \leftarrow$ MaxSumSubseq($m+1$,$n$)
    $\mathsf{subsol}_l \leftarrow 0$
    for $i \leftarrow m \ldots s$ do
        if $\sum_{k=i}^{m} a_k > \mathsf{subsol}_l$ then
            $\mathsf{subsol}_l \leftarrow \sum_{k=i}^{m} a_k$
        end if
    end for
    $\mathsf{subsol}_r \leftarrow 0$
    for $i \leftarrow m+1 \ldots t$ do
        if $\sum_{k=m+1}^{i} a_k > \mathsf{subsol}_r$ then
            $\mathsf{subsol}_r \leftarrow \sum_{k=m+1}^{i} a_k$
        end if
    end for
    return $\max(\mathsf{sol}_l, \mathsf{sol}_r, \mathsf{subsol}_l + \mathsf{subsol}_r)$
end function
return MaxSumSubseq($1$,$n$)
```

Can we do better? Suppose someone tells us that the optimal solution has to contain the boundary between a_m and a_{m+1} for some m. Then, we could immediately perform the 'merging step' of Algorithm 5 and find the optimal solution in $O(n)$ time. However, of course, the problem is that no one tells us where m is, but we can try all possible values of m: Algorithm 6 shows the approach.

Algorithm 6 Fourth attempt (quadratic algorithm)

```
for $m = 1, \ldots, n-1$ do
    $\mathsf{subsol}^l_m = \max\left[\{\sum_{k=i}^{m} a_k \mid 1 \le i \le m\} \cup \{0\}\right]$
    $\mathsf{subsol}^r_{m+1} = \max\left[\{\sum_{k=m+1}^{i} a_k \mid m+1 \le i \le n\} \cup \{0\}\right]$
end for
$\mathsf{sol} \leftarrow 0$
for $m = 1, \ldots, n-1$ do
    if $\mathsf{subsol}^l_m + \mathsf{subsol}^r_{m+1} > \mathsf{sol}$ then
        $\mathsf{sol} \leftarrow \mathsf{subsol}^l_m + \mathsf{subsol}^r_{m+1}$
    end if
end for
return $\mathsf{sol}$
```

Unfortunately, the running time of Algorithm 6, implemented somewhat naïvely, is $O(n^2)$. However, the following observation enables implementing Algorithm 6 to run in $O(n)$ time.

Observation 1. *For $m > 1$,* $\mathsf{subsol}^l_m = \max\left(\mathsf{subsol}^l_{m-1} + a_m, 0\right)$.

Proof. If subsol^l_m is nonzero, the contiguous subsequence yielding subsol^l_m has to be obtained from the maximum subsequence ending at a_{m-1} plus a_m. □

In fact, "someone" can as well tell us exactly where the optimal subsequence ends; this leads to the $O(n)$-time algorithm given by Algorithm 7.

Algorithm 7 Fourth attempt

```
1:  $\mathsf{MaxEndingAt}_1 = \max(a_1, 0)$
2:  for $m = 2, \ldots, n$ do
3:      $\mathsf{MaxEndingAt}_m = \max\left(\mathsf{MaxEndingAt}_{m-1} + a_m, 0\right)$
4:  end for
5:  $\mathsf{sol} \leftarrow 0$
6:  for $m = 1, \ldots, n$ do
7:      if $\mathsf{MaxEndingAt}_m > \mathsf{sol}$ then
8:          $\mathsf{sol} \leftarrow \mathsf{MaxEndingAt}_m$
9:      end if
10: end for
11: return $\mathsf{sol}$
```

Can we do better? No.

Theorem 4. *There is no deterministic algorithm for Problem 2 that runs in sublinear time.*

Proof sketch. Suppose there is one. Then, for some sufficiently large n_0, there exists an input $a_1, \ldots, a_{n_0}$ for which the algorithm does not access the value of a_k for some $1 \leq k \leq n_0$.
If the algorithm finds the subsequence that contains a_k, the algorithm would produce an incorrect output for $a_1, \ldots, a_{k-1}, -\infty, a_{k+1}, \ldots, a_{n_0}$.
If the algorithm finds the subsequence that does not contain a_k, the algorithm would produce an incorrect output for $a_1, \ldots, a_{k-1}, +\infty, a_{k+1}, \ldots, a_{n_0}$. □

References

[1] J. Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, Sept. 1984.