COMP 3711 Design and Analysis of Algorithms

Tutorial: Dynamic Programming

Decoding Numbers to Letters

Decoding Numbers to Letters

1) "A" → 1, "B" → 2, . . ., "Z" → 26

Given an encoded message *A* containing *n* digits in 1-9, design a *O(n)* time dynamic programming algorithm to determine the total number of ways to decode *A*.

Example:

15243 could be decoded 4 different ways as

1 5 2 4 3 = A E B D C

1 5 24 3 = A E X C

15 2 4 3 = O B D C

15 24 3 = O X C

Decoding Numbers to Letters

1) "A" → 1, "B" → 2, . . ., "Z" → 26

Given an encoded message *A* containing *n* digits in 1-9, design a *O(n)* time dynamic programming algorithm to determine the total number of ways to decode *A*.

Let $d[i]$ be the total number of ways to decode A[1..i].

Base Cases:

$d[1] = 1$, since there is only one way to decode the items

Working through the possibilities (checking whether A[1,2] can encode a single letter or not, shows

$$d[2] = \begin{cases} 1 & \text{if } 10 \ * A[i-1] + A[i] > 26 \\ 2 & \text{otherwise} \end{cases}$$

Decoding Numbers to Letters

1) "A" → 1, "B" → 2, . . ., "Z" → 26

Given an encoded message *A* containing *n* digits in 1-9, design a *O(n)* time dynamic programming algorithm to determine the total number of ways to decode *A*.

Let $d[i]$ be the total number of ways to decode A[1..i].

General Case: If $\mathrm{i} > 2$,

If $\mathrm{A[i-1, i]}$ can not encode a letter *(because it is > 26)* then the decoding must have A[i] encoding a unique letter. Otherwise there are two possibilities: either A[i-1, i] encodes a single letter or encodes two different letters. This yields

$$d[i] = \begin{cases} d[i-1] & \text{if } 10 * A[i-1] + A[i] > 26 \\ d[i-2] + d[i-1] & \text{otherwise} \end{cases}$$

Decoding Numbers to Letters

1) “A” → 1, “B” → 2, . . ., “Z” → 26

Given an encoded message *A* containing *n* digits in 1-9, design a *O(n)* time dynamic programming algorithm to determine the total number of ways to decode *A*.

Let $d[i]$ be the total number of ways to decode A[1..i])

Base Case: $d[1] = 1$ $d[2] = \begin{cases} 1 & \text{if } 10 * A[1] + A[2] > 26 \\ 2 & \text{otherwise} \end{cases}$

General Case: If $i > 2$,

$$d[i] = \begin{cases} d[i-1] & \text{if } 10 * A[i-1] + A[i] > 26 \\ d[i-2] + d[i-1] & \text{otherwise} \end{cases}$$

This can be implemented in $O(n)$time.
$O(1)$ time to calculate each d[i]; $O(n)$ time to calculate all the d[i].

Longest Monotonically Increasing Subsequence

Question 1

Give an $O(n^2)$ time dynamic programming algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers, i.e, each successive number in the subsequence is greater than or equal to its predecessor.

For example, if the input sequence is

$$\langle 5, 24, 8, 17, 12, 45 \rangle,$$

the output should be either $\langle 5, 8, 12, 45 \rangle$ or $\langle 5, 8, 17, 45 \rangle$.

Solution

We first give an algorithm which finds the **length** of the longest increasing subsequence; will later modify it to report a subsequence with this length.

Let $X_i = \langle x_1, \cdots, x_i \rangle$ denote the prefix of X consisting of its first i items.

Define

$c[i]$ = the length of the longest increasing subsequence that **ends** at x_i.

The length of the longest increasing subsequence in X is then

$$\max_{1 \le i \le n} c[i].$$

Solution

$c[i]$ = the length of the longest increasing subsequence that **ends** at x_i.

Initial Condition: $\mathrm{c}[1] = 1$

If $i > 1$:

If all items to left of x_i are $>$ than x_i, answer must be 1.

Otherwise, longest increasing subsequence **that ends with x_i** has form $\langle Z, x_i \rangle$, where Z is the longest increasing subsequence **that ends with x_r** for some $r < i$ and $x_r \leq x_i$.

This yields the following recurrence relation:

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \leq r < i \\ \max\limits_{\substack{1 \leq r < i \\ x_r \leq x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Solution

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

We do not write the pseudocode, but just note that we store the $c[i]$'s in an array whose entries are computed in order of increasing i .

After computing the c array, we run through all the entries to find the maximum value.

This is the length of the longest increasing subsequence in X.

For every i it takes $O(i)$ time to calculate c_i.

=> the running time is $O(\sum_{i=1}^{n} i) = O(n^2)$.

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is $X = \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1					

Solution:

$i = 1$: $c[1] = 1$

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \leq r < i \\ \max\limits_{\substack{1 \leq r < i \\ x_r \leq x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is X $= \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2				

Solution:

$i = 1$: $c[1] = 1$

$i = 2$: Since $x_1 \leq x_2 \Rightarrow c[2] = \max\{c[1]\} + 1 = 2$

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is $X = \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3			

Solution:

$i = 1$: $c[1] = 1$

$i = 2$: Since $x_1 \le x_2 \;\Rightarrow\; c[2] = \max\{c[1]\} + 1 = 2$

$i = 3$: Since $x_1, x_2 \le x_3 \Rightarrow \mathrm{c}[3] = \max\{c[1], c[\,2]\} + 1 = 2 + 1 = 3$

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is $X = \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3	1		

Solution:

$i = 1$: $c[1] = 1$

$i = 2$: Since $x_1 \le x_2 \quad \Rightarrow \quad c[2] = \max\{c[1]\} + 1 = 2$

$i = 3$: Since $x_1, x_2 \le x_3 \Rightarrow \mathrm{c}[3] = \max\{c[1], c[2]\} + 1 = 2 + 1 = 3$

$i = 4$: Since $x_1, x_2, x_3 > x_4 \Rightarrow \mathrm{c}[4] = 1$

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is $X = \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3	1	2	

Solution:

$i = 1$: $c[1] = 1$

$i = 2$: Since $x_1 \le x_2 \ \Rightarrow \ c[2] = \max\{c[1]\} + 1 = 2$

$i = 3$: Since $x_1, x_2 \le x_3 \Rightarrow \mathrm{c}[3] = \max\{c[1], c[2]\} + 1 = 2 + 1 = 3$

$i = 4$: Since $x_1, x_2, x_3 > x_4 \ \Rightarrow \ \mathrm{c}[4] = 1$

$i = 5$: Since $x_4 \le x_5$ and $x_1, x_2, x_3 > x_5 \ \Rightarrow \mathrm{c}[5] = \max\{c[4]\} + 1 = 2$

example

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

Question:

The input sequence is $X = \{4, 5, 7, 1, 3, 9\}$;
Find the longest monotonically increasing subsequence.

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3	1	2	4

Solution:

Return: max is $\text{c}[6] = 4$

$i = 1$: $c[1] = 1$

$i = 2$: Since $x_1 \le x_2 \Rightarrow c[2] = \max\{c[1]\} + 1 = 2$

$i = 3$: Since $x_1, x_2 \le x_3 \Rightarrow \text{c}[3] = \max\{c[1], c[2]\} + 1 = 2 + 1 = 3$

$i = 4$: Since $x_1, x_2, x_3 > x_4 \Rightarrow \text{c}[4] = 1$

$i = 5$: Since $x_4 \le x_5$ and $x_1, x_2, x_3 > x_5 \Rightarrow \text{c}[5] = \max\{c[4]\} + 1 = 2$

$i = 6$: Since $x_1, x_2, x_3, x_4, x_5 \le x_6 \Rightarrow \text{c}[6] = \max\{c[1], c[2], c[3], c[4], c[5]\} + 1 = 4$

Solution

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

To report optimal subsequence, we need to store for each i, not only $c[i]$, but also value of r which achieves the maximum in the recurrence relation.

Denote this by $r[i]$. (Ø means no predecessor)

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3	1	2	4
r[i]	Ø	1	2	Ø	4	3

Suppose $c[k] = \max\limits_{1 \le i \le n} c[i]$. Let S be optimal subsequence

x_k is the last item in S. the optimal subsequence.

2nd to last item in S is $x_{r[k]}$,

3rd to last item in S is $x_{r[r[k]]}$, etc.

until we have found all the items in S

Running time of this step is $O(n)$, so entire algorithm is still $O(n^2)$.

Solution

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for all } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{other cases} \end{cases}$$

To report optimal subsequence, we need to store for each i, not only $c[i]$, but also value of r which achieves the maximum in the recurrence relation.

Denote this by $r[i]$. (∅ means no predecessor)

i	1	2	3	4	5	6
X	4	5	7	1	3	9
c[i]	1	2	3	1	2	4
r[i]	∅	1	2	∅	4	3

Return: max is $\mathrm{c}[6] = 4$, so $k = 6$

$r[6] = 3$

$r[r[6]] = r[3] = 2$

$r[r[r[6]]] = r[2] = 1$

$r[r[r[r[6]]]] = r[1] = \emptyset$

Solution is

$x_{r[r[r[6]]]} \leftarrow x_{r[r[6]]} \leftarrow x_{r[6]} \leftarrow x_6$

i.e. $x_1 \leftarrow x_2 \leftarrow x_3 \leftarrow x_6$

i.e. {4, 5, 7, 9}

The longest oscillating subsequence problem

The longest oscillating subsequence problem

A sequence of numbers $a_1, a_2, \ldots a_n$ is *oscillating* if

$$a_i < a_{i+1} \text{ for every odd index } i$$

and

$$a_i > a_{i+1} \text{ for even index } i$$

For example, the sequence below is oscillating.

2, 7, 1, 8, 2, 6, 1, 8, 3

Describe and analyze an efficient algorithm to find a longest oscillating subsequence in a sequence of *n* integers.

Your algorithm only needs to output the **length** of the oscillating subsequence.

For example if the input sequence is 2, 4, 5, 1, 4, 2, 1, your algorithm should output 5, corresponding to the subsequence **2, 4, 1, 4, 1**, or **2, 4, 1, 4, 2**, or any other such subsequence.

For full credit, your algorithm should run in $O(n^2)$ time.

The longest oscillating subsequence problem

Uses similar idea to longest increasing subsequence.

Let $o[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **odd** length;

Let $e[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **even** length;

Base Case: $o[1] = 1$; $e[1] = -\infty$.

Main New Observation:
In order to be able to add a new item to the end of an oscillating sequence that ended at a previous a_j we need to know if that sequence was odd size (went down) or even size (went up).

That requires maintaining TWO different tables.
One for odd size oscillating subseqs and one for even ones.

The longest oscillating subsequence problem

Uses similar idea.

Let $o[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **odd** length;

Let $e[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **even** length;

General Case: For $o[i]$

The longest odd oscillating sequence ending with a_i is either
a_i by itself or
$< Z, a_i >$ where Z is an even oscillating sequence ending at some a_j where $a_j > a_i$ and $j < i$.

==> $$\boxed{o[i] = 1 + \max_{j<i \;\&\; a_j>a_i} \{0, e[j]\}}$$

The longest oscillating subsequence problem

Uses similar idea.

Let $o[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **odd** length;

Let e$[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **even** length;

General Case: For e$[i]$

If for all $j < i,\ a_j > a_i$

=> no even oscillating subsequence ending at a_i exists.

Otherwise, longest even oscillating sequence ending with a_i is $< Z, a_i >$ where Z is an odd oscillating sequence ending at a_j where $a_j < a_i$ and $j < i$.

==> $$\boxed{e[i] = \begin{cases} -\infty & \text{if } \ a_j > a_i \text{ for all } j < i \\ 1 + \max\limits_{j<i \ \& \ a_j<a_i} \{o[j]\} & \text{otherwise} \end{cases}}$$

The longest oscillating subsequence problem

Uses similar idea.

Let $o[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **odd** length;

Let e$[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **even** length;

General Case:

$$\boxed{\mathrm{o}[i] = 1 + \max_{j<i \ \& \ a_j>a_i} \{0, e[j]\}}$$

$$e[i] = \begin{cases} -\infty & \text{if} \quad a_j > a_i \text{ for all } j < i \\ 1 + \max_{j<i \ \& \ a_j<a_i} \{o[j]\} & \text{otherwise} \end{cases}$$

Can then calculate the values in the tables in order

$$o[1], e[1], o[2], e[2], o[3], e[3], \dots .$$

The longest oscillating subsequence problem

Let $o[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **odd** length;
Let e$[i]$ be the length of the longest oscillating subsequence that ends at a_i and has an **even** length;

Base Case: $o[1] = 1$; $e[1] = -\infty$.

General Case

$$\mathrm{o}[i] = 1 + \max_{j<i \ \& \ a_j > a_i} \{0, e[j]\}$$

$$e[i] = \begin{cases} -\infty & \text{if } a_j > a_i \text{ for all } j < i \\ 1 + \max_{j<i \ \& \ a_j < a_i} \{o[j]\} & \text{otherwise} \end{cases}$$

Final solution is maximum of all the $o[i], e[i]$

Since each $\mathrm{o}[i]$ and $\mathrm{e}[i]$ can be calculated in $O(n)$ time, entire algorithm requires $O(n^2)$ time.

DP Maximum Contiguous Subarray

The Maximum Subarray Problem: A DP solution

Input: Profit history of a company. Money earned/lost each year.

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	3	2	1	-7	5	2	-1	3	-1

Problem: Find the span of years in which the company earned the most

Answer: Year 5-8 , 9 M$

Formal definition:

Input: An array of numbers $A[1 \dots n]$, both positive and negative

Output: Find the maximum value $V(k, i)$, where $V(k, i) = \sum_{t=k}^{i} A[t]$

Recall

Previously learnt 4 different algorithms for solving this problem

- $\Theta(n^3)$ Brute force Algorithm
- $\Theta(n^2)$ (Reuse of Information) Algorithm
- $\Theta(n \log n)$ Divide-and-Conquer Algorithm
- $\Theta(n)$ Linear Scan Algorithm

- Now: design a $\Theta(n)$ Dynamic Programming Algorithm

Note: previous algorithms solved a slightly different problem than the one defined on the previous page. The problems differ (ONLY) in the case that ***for all*** $\boldsymbol{i}$, $\boldsymbol{A[i] < 0}$.

In that case, the old algorithms returned the value 0.
The problem as defined on the previous page returns $\max_i A[i]$.

Easy to transform the solution of one problem to that of the other in $\Theta(n)$ *time.*

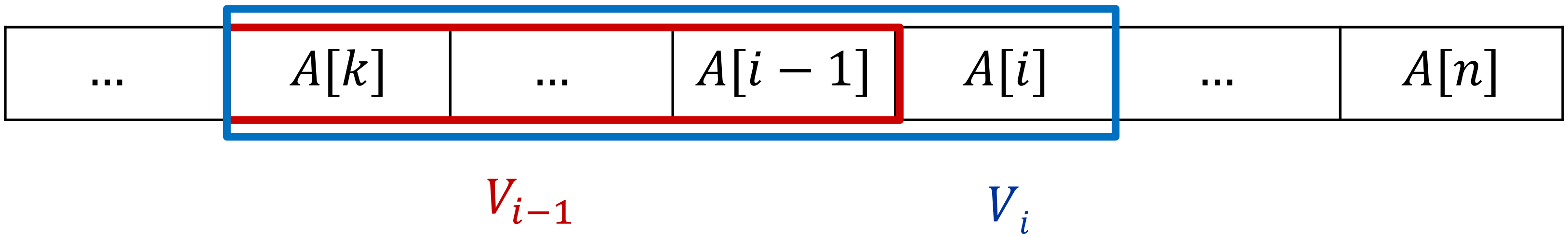
A dynamic programming ($\Theta(n)$) algorithm

Define: V_i to be max value subarray ending at $A[i]$

$$V_i = \max_{1 \le k \le i} V(k, i)$$

The main observation is that if $V_i \neq A[i] = V(i, i)$ then

$$V_i = A[i] + \max_{1 \le k < i} V(k, i-1) = A[i] + V_{i-1}$$



This immediately implies DP Recurrence

$$\boxed{V_i = \begin{cases} A[1] & \text{if } i = 1 \\ \max\{A[i], A[i] + V_{i-1}\} & \text{if } i > 1 \end{cases}}$$

The DP recurrence

Set $V_i = \max_{1 \le k \le i} V(k,i)$. We just saw

$$V_i = \begin{cases} A[1] & \text{if } i = 1 \\ \max\{A[i], A[i] + V_{i-1}\} & \text{if } i > 1 \end{cases}$$

Original problem then becomes finding i' such that

$$V_{i'} = \max_{1 \le i \le n} V_i$$

The DP recurrence permits constructing V_i in O(1) time from V_{i-1}.

$\Rightarrow$ We can construct $V_1, V_2, \ldots, V_n$ in order in O(n) total time while keeping track of the largest V_i found so far

$\Rightarrow$ This finds $V_{i'}$ in O(n) total time, solving the problem.

Note: This algorithm is very similar to the linear scan algorithm we developed in class, but found using DP reasoning

Implementation

Derived recurrence that

$$\boxed{V_i = \begin{cases} A[1] & \text{if } i = 1 \\ \max\{A[i], A[i] + V_{i-1}\} & \text{if } i > 1 \end{cases}}$$

where

$$\boxed{V_i = \max_{1 \le k \le i} V(k,i)}$$

and need to find i' such that

$$\boxed{V_{i'} = \max_{1 \le i \le n} V_i}$$

This is very straightforward.
Next slides give actual code, and a worked example

Version 1

Store V_i in a table $V[1,2,\dots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$

Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i−1])
      if V_max < V[i]
              then V_max ← V[i]
      end if
return V_max
```

Running time: $\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7	6	9	8
V_{max}	3	5	6	6	6	7	7	9	9

Solution is $V[8]$

Version 2

Simplified: We only need to remember the last V_i (call it V) and V_{max}

Base condition: $V \leftarrow A[1]$

Recurrence: $V \leftarrow \max(A[i], A[i] + V)$

```
V ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V ← max(A[i], A[i] + V)
      if V_max < V
            then V_max ← V
      end if
return V_max
```

Running time: $\Theta(n)$

This gets same result as Version 1, but is simpler!

Next pages provide a detailed walk-through of how Version 1 fills in the DP table.

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$

Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
            then V_max ← V[i]
      end if
return V_max
```

Running time: $\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3								
V_{max}	3								

$V_{max} = V[1] = A[1] = 3$

Version 1

Store V_i in a table $V[1, 2, \dots, n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,…,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
        V[i] ← max(A[i], A[i] + V[i − 1])
        if V_max < V[i]
                then V_max ← V[i]
        end if
return V_max
```

Running time: $\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5							
V_{max}	3	5							

$V_{max} = \max(A[2], A[2] + V[1]) = \max(2, 2 + 3) = 5$

Version 1

Store V_i in a table $V[1, 2, \dots, n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,…,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
                then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6						
V_{max}	3	5	6						

$V_{max} = \max(A[3], A[3] + V[2]) = \max(1, 1 + 5) = 6$

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$

Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
            then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1					
V_{max}	3	5	6	6					

$V_{max} = 6 > \max(A[4], A[4] + V[3]) = \max(-7, -7 + 6) = -1$

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$

Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
              then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5				
V_{max}	3	5	6	6	6				

$V_{max} = 6 > \max(A[5], A[5] + V[4]) = \max(5, 5-1) = 5$

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
              then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7			
V_{max}	3	5	6	6	6	7			

$V_{max} = \max(A[6], A[6] + V[5]) = \max(2, 2+5) = 7$

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
                then V_max ← V[i]
      end if
return V_max
```

Running time: $\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7	6		
V_{max}	3	5	6	6	6	7	7		

$V_{max} = 7 > \max(A[7], A[7] + V[6]) = \max(-1, -1 + 7) = 6$

Version 1

Store V_i in a table $V[1, 2, \ldots, n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
             then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7	6	9	
V_{max}	3	5	6	6	6	7	7	9	

$V_{max} = \max(A[8], A[8] + V[7]) = \max(3, 3 + 6) = 9$

Version 1

Store V_i in a table $V[1,2,\dots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
        V[i] ← max(A[i], A[i] + V[i − 1])
        if V_max < V[i]
                then V_max ← V[i]
        end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7	6	9	8
V_{max}	3	5	6	6	6	7	7	9	9

$V_{max} = 9 > \max(A[9], A[9] + V[8]) = \max(-1, -1 + 9) = 8$

Version 1

Store V_i in a table $V[1,2,\ldots,n]$, at each step calculating $V[i]$ from $V[i-1]$

Base condition: $V[1] \leftarrow A[1]$ Recurrence: $V[i] \leftarrow \max(A[i], A[i] + V[i-1])$

```
let V[1,2,...,n] be an array storing V_i
V[1] ← A[1]
V_max ← A[1]
for i ← 2 to n do
      V[i] ← max(A[i], A[i] + V[i − 1])
      if V_max < V[i]
               then V_max ← V[i]
      end if
return V_max
```

Running time:
$\Theta(n)$

i	1	2	3	4	5	6	7	8	9
$A[i]$	3	2	1	-7	5	2	-1	3	-1
$V[i]$	3	5	6	-1	5	7	6	**9**	8
V_{max}	3	5	6	6	6	7	7	9	9

Solution is $V[8]$

$V_{max} = 9 > \max(A[9], A[9] + V[8]) = \max(-1, -1 + 9) = 8$

COMP3711

Number of contiguous subarrays with average k

Number of contiguous subarrays with average k

Describe and analyze an efficient algorithm to find the number of contiguous subarrays from an array $A[1..n]$ that have an average equal to k.

For example, if the input array is 1, 3, 1, 5, 7 and $k = 3$, your algorithm should output 3, corresponding to the subarrays **{3}, {1,5}, {3,1,5}**.

For full credit, your algorithm should run in $O(n)$ time.

Solution

Describe and analyze an efficient algorithm to find the number of contiguous subarrays from an array $\boldsymbol{A[1..n]}$ that have an average equal to $\boldsymbol{k}$.

Note: Given an array $A[1..n]$ with n elements, if the average is avg(A), then

$$\text{avg}(A) = \frac{A[1] + A[2] + \cdots + A[n]}{n}$$
$$\text{avg}(A) \times n = A[1] + A[2] + \cdots + A[n]$$
$$0 = (A[1] - \text{avg}(A)) + (A[2] - \text{avg}(A)) + \cdots + (A[n] - \text{avg}(A))$$

If we subtract k from every element of A and the sum is equal to 0 $\sum_{\forall i}(A[i] - k) = 0$, then the average of all the elements is k (avg$(A) = k$).

If we split A to two subarrays at a random index j such that $A[1..j]$, $A[j+1..n]$, and $\sum_{\forall i}(A[i] - k) = c$, then

$$\sum_{i=j+1}^{n}(A[i] - k) = c - \sum_{i=1}^{j}(A[i] - k).$$

Solution

Describe and analyze an efficient algorithm to find the number of contiguous subarrays from an array $\boldsymbol{A[1..n]}$ that have an average equal to $\boldsymbol{k}$.

If $\sum_{\forall i}(A[i] - k) = 0$, then $\text{avg}(A) = k$. (1)

If we split A to two subarrays at a random index j such that $A[1..j]$, $A[j+1..n]$, and $\sum_{\forall i}(A[i] - k) = c$, then

$$\sum_{i=j+1}^{n}(A[i] - k) = c - \sum_{i=1}^{j}(A[i] - k). \qquad (2)$$

We name $A[1..j]$ as the *prefix* and $A[j+1..n]$ as the *suffix*.

The *suffix* $A[j+1..n]$ always includes at least the last element, i.e., $A[n]$.

To find all the *suffixes* $A[j+1..n]$ that end with $A[n]$ and have an average of k, we count all the *prefixes* such that $\sum_{i=1}^{j}(A[i] - k) = c$ (by equation 1 & 2).

Thus, we sum and count the contiguous subarrays that end at $A[1], A[2], \dots, A[n]$.

Naïve Solution

Describe and analyze an efficient algorithm to find the number of contiguous subarrays from an array $\boldsymbol{A[1..n]}$ that have an average equal to $\boldsymbol{k}$.

To find all the *suffixes* that end with $A[n]$ and have an average of k, we find all the *prefixes* such that $\sum_{i=1}^{j}(A[i]-k)=c$ (by equation 1 & 2).

Thus, we sum and count the contiguous subarrays that end at $A[1], A[2], \dots, A[n]$.

Let $S_j = \sum_{i=0}^{j}(A[i]-k)$ for $0 \le j < n$, we store S_j in a table S.

For any j and $S_j = c$, we find all *prefixes* such that $S_i = c$ and $0 \le i < j$. We count all such occurrences and add them to r_j.

r_j stores the count of the contiguous subarrays that satisfy the condition up to j.

The base cases are $S_0 = 0$ and $r_0 = 0$.

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = r_{i-1} + (\# \text{ of } S_i \text{ in } S[0..i-1])$$

Let $A = d[1,3,1,5,7]$ and $k = 3$ we set up our base base cases:

$S_0 = 0$

i	0	1	2	3	4	5
S	0					

$r_0 = 0$

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$r_i = r_{i-1} + (\# \text{ of } S_i \text{ in } S[0..i-1])$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_1 = S_0 + 1 - 3 = -2$

$-2 \notin S[0..0]$ so,

$r_1 = r_0$

i	0	1	2	3	4	5
S	0	-2				

We update

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = r_{i-1} + (\# \text{ of } S_i \text{ in } S[0..i-1])$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_2 = S_1 + 3 - 3 = -2$

$-2 = S_1$ so,

$r_2 = r_1 + 1 = 1$

This implies that we remove the *prefix* $[1]$

i	0	1	2	3	4	5
S	0	-2	-2			

We update

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$r_i = r_{i-1} + (\#\text{ of } S_i \text{ in } S[0..i-1])$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_3 = S_2 + 1 - 3 = -4$

$-4 \notin S[0..2]$ so,

$r_3 = r_2 = 1$

i	0	1	2	3	4	5
S	0	-2	-2	-4		

We update

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = r_{i-1} + (\# \text{ of } S_i \text{ in } S[0..i-1])$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_4 = S_3 + 5 - 3 = -2$

$-2 = S_1$ and $-2 = S_2$ so,

$r_4 = r_3 + 2 = 3$

This implies that we remove the *prefixes* $[1]$ and $[1,3]$

i	0	1	2	3	4	5
S	0	-2	-2	-4	-2	

We update

Naïve Solution

Base Case: $S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = r_{i-1} + (\# \text{ of } S_i \text{ in } S[0..i-1])$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_5 = S_4 + 7 - 3 = 2$

$2 \notin S[0..4]$ so,

$r_5 = r_4 = 3$

i	0	1	2	3	4	5
S	0	-2	-2	-4	-2	2

We update

Finally, we return r_5

Solution

Describe and analyze an efficient algorithm to find the number of contiguous subarrays from an array $\boldsymbol{A[1..n]}$ that have an average equal to $\boldsymbol{k}$.

Instead of using a table S, we use a hash map d to store the number of occurrences of each S_i.

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$, we set up our base cases:

Stored values on hash map d

$S_0 = 0$

$d[0] = 1$

$r_0 = 0$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$$\uparrow$$

$S_1 = S_0 + 1 - 3 = -2$

$S_1 \notin d$ so,

$r_1 = r_0$

Stored values on hash map d

$d[0] = 1$

We update $\longrightarrow$ $d[-2] = 1$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_2 = S_1 + 3 - 3 = -2$

$S_2 \in d$ so,

$r_2 = r_1 + d[S_2] = 1$

This implies that we remove the *prefix* $[1]$

Stored values on hash map d

$d[0] = 1$

$d[-2] = 1$

We update $\longrightarrow$ $d[-2] = 2$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_3 = S_2 + 1 - 3 = -4$

$S_3 \notin d$ so,

$r_3 = r_2 = 1$

Stored values on hash map d

$d[0] = 1$

$d[-2] = 2$

We update $\longrightarrow$ $d[-4] = 1$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_4 = S_3 + 5 - 3 = -2$

$S_4 \in d$ so,

$r_4 = r_3 + d[S_4] = 3$

This implies that we remove the *prefixes* [1] and [1,3]

Stored values on hash map d

$d[0] = 1$

$d[-2] = 2$

$d[-4] = 1$

We update $\longrightarrow$ $d[-2] = 3$

Solution

Base Case: $d[0] = 1, S_0 = 0$ and $r_0 = 0$ for $i = 0$

General Case: If $i > 0$

$$r_i = \begin{cases} r_{i-1} + d[S_i] & \text{if } S_i \in d \\ r_{i-1} & \text{otherwise} \end{cases}$$

$$d[S_i] \mathrel{+}= 1$$

Let $A = d[1,3,1,5,7]$ and $k = 3$

$S_5 = S_4 + 7 - 3 = 2$

$S_5 \notin d$ so,

$r_5 = r_4 = 3$

Stored values on hash map d

$d[0] = 1$

$d[-2] = 3$

$d[-4] = 1$

We update $\longrightarrow$ $d[2] = 1$

Finally, we return r_5