# Intro to Algorithms HW 8

Greg Stewart

April 20, 2018

## Q6.1 Give a linear time algorithm for the following task:

*Take a list of numbers $a_1, a_2, \ldots, a_n$. Output the contiguous subsequence of maximum sum.*

To do this, we need to keep track of a starting index, $s$, ending index, $e$, a useful index-tracking variable $i$, the maximum, which we initialize to a very small number (i.e. very negative), and a tracking value which starts at 0. Iterate through the array and do the following at each iteration.

To the tracking value, add the value of the array at the current index. If the tracking value is less than zero, set it to 0 and set the tracking variable $i$ to the current index of the array plus 1. If the maximum is less than the tracking value, set the maximum equal to the tracking value, $s$ equal to $i$, and $e$ equal to the current index of the array.

Once this has completed, the sum can be printed from the maximum, and the sequence of the array can be printed by printing the elements from $s$ to $e$.

```
function:
  s := 0
  e := 0
  i := 0
  max := -inf
  track := 0

  for j := 1..n
    track := track + a[j]

    if track < 0
      track := 0
      i := j + 1

    if max < track
      max := track
      s = i
      e = j

  for j := s..e
    print a[j]
```

## Q6.4 Given a string of $n$ characters believed to be e corrupted text document where punctuation has vanished, you wish to reconstruct the document using a dictionary, available as such:

$$dict(w) = \begin{cases} \text{true} & \text{if } w \text{ is a valid word} \\ \text{false} & \text{otherwise} \end{cases}$$

(a) *Give a dynamic programming algorithm to determine whether the string s[] can be reconstituted as a sequence of valid words, with running time of at most $O(n^2)$, assuming calls to dict(w) take unit time.*

The subproblems seem clear here. $s[1 \ldots j]$ is only valid if $s[1 \ldots i]$ is a real sequence of words and $s[i+1 \ldots j]$ is a real word. Now rewrite valid(j) as the following:

$$valid(j) = \begin{cases} \text{false} & \text{if } i \leq 0 \\ \max_{1 \leq i < j}(valid[i] \wedge dict(s[i+1 \ldots j])) & \text{otherwise} \end{cases}$$

Now, valid(j) depends only on valid(i) when i ¡ j, so we can solve valid(1), valid(2), ..., valid(n) with the following algorithm:

Let valid[1...n] be a boolean array. Iterate $j$ from 1 to $n$. On each iteration, do the following. Set valid[j] to false. Iterate $i$ from 1 to $j-1$, and if valid[i] and dict[i+1...j] are both true, then assign valid[j] to be true.

```
function:
  valid[1..n] is a bool array

  for j := 1..n
    valid[j] := false
    for i := 1..j-1
      if valid[i] and dict[i+1..j]
        valid[j] := true

  if valid[n]
    return true
```

(b) *In the event that the string is valid, make your algorithm output the corresponding sequence of words.*

This just needs to keep track of the appropriate indices. Add an array, says words[1...n], to the algorithm. When iterating $j$, assign words[j] to 0. Then when iterating $i$, if the if statement is successful, add an assignment: set words[j] = i. To output the words afterwards, set some variable $x$ to $n$, and while $x > 0$, add the word s[(words[x]+1)...x] to a stack, then once that's done, pop words off the stack and print them until it's empty.

```
function:
  valid[1..n] is a bool array
  words[1..n] is an array of integers
  for j := 1..n
    valid[j] := false
    words[j] := 0
    for i := 1..j-1
      if valid[i] and dict[i+1..j]
        valid[j] := true
        words[j] := i

  if valid[n]
    x := n
    initialize stack
    while x > 0
      add s[(words[x]+1)..x] to stack
      x -= (x - (words[x] + 1))

    while stack not empty
      pop word from stack and print
```

**Q6.17 Given an unlimited supply of coins with denominations $x_i$, we wish to make change to a value $v$. This might not be possible. Give an $O(nv)$ dynamic programming algorithm for the following:**

*Take $x_1, \ldots, x_n$ and $v$. Answer whether it's possible to make change for $v$ using the coins given.*

The subproblems needed here are fairly straightforward. Let $C(v)$ be true if it's possible to make change for $v$. Then it is possible to make change for $v - x_i$ for each $x_i$. This must be done for all $x_i$, however.

$$C(v) = \begin{cases} \text{true} & C(v - x_i) \text{ for } x_i \leq v \\ \text{false} & \text{otherwise} \end{cases}$$

This leads to the following algorithm. Let there be an array C[] of size $v + 1$, and let $C[0]$ be trye. Set all other values in the array to false. Iterate $i$ from 1 to $v$, and upon each iteration, iterate $j$ from 1 to $n$. Within this iteration, if $x_j \leq v$, set C[i] to true if either C[i] or C[i - $x_j$] is true. Otherwise, set C[i] to false. Once the loops are complete, output C[v].

```
function:
  C[0..v] is an array of boolean values
  C[0] := true
  for i := 1..v
    C[i] := false

  for i := 1..v
    for j := 1..n
      if x_j <= v
        if C[i] or C[i-x[j]]
          C[i] := true
        else
          C[i] := false

  return C[v]
```

This is $O(nv)$ because the outer loop is from 1 to $v$ and the nested loop is from 1 to $n$, so they are multiplied together, and the operations within are all unit time.

## Q6.19 Given another coin set, we want to make change for $v$, but this time using at most $k$ coins. May not be possible. So do this:

*Take $x_1, \ldots, x_n$, $k$, and $v$. Answer whether it's possible to make change for $v$ using $k$ coins.*

Define $C(y) = c$, where $c$ is the smallest number of coins needed to get the total value $y$, and where $1 \leq y \leq v$. To split this into subproblems, we can use a **recurrence**: $C(y)$ is equal to the minimum of $C(y - x_i) + 1$, over all the $x_i$.

Let $C(0) = 0$, and let $C = \infty$ for all negative values. Iterate from 1 to $v$. On each iteration, perform the recurrence using the previous values of $C$. Once this is done, if $C(v) \leq k$, then it's possible to make change with $k$ or fewer coins. Otherwise, it's not.

```
function:
  C[0..v] is an array of integers
  C[0] := 0

  for i := 1..v
    min = inf
    for j := 1..n
      if i-x[j] >= 0
        tmp := C[i-x[j]] + 1
        if tmp < min
          min := tmp
    C[i] := min

  if C[v] <= k
    return true

  return false
```

The running time is $O(nv)$ as all of the $x_i$ from 1 to $n$ must be considered for each iteration from 1 to $v$.