Complete recap so far

Serge Kruk

September 29, 2021

My favourite section

We will solve the same problem by crafting five different algorithms, improving the runtime at each step, by looking at it in different ways.

Design Of Algorithms

- This is possibly the most important section of the class!
- It certainly is the most important so far, as it reviews everything and contextualizes.

Problem statement (Max contiguous sum)

Given an array a of real numbers, find a contiguous subarray with the largest possible sum. Define the largest sum of an empty array to be zero. Note that this implies that an array with only negative numbers will have a max contiguous sum of zero.

$$\underbrace{\underbrace{31, -41}_{-10}, 59, 26, -53, 58, 97}_{49}, -93, -23, 84$$

$$31, -41, \underbrace{59, 26, -53, 58, 97}_{187}, -93, -23, 84$$

Discuss! (First, find brute-force approach.)

Brute force solution

Consider every subarray using two loops, one for the beginning and one for the end. Sum each subarray and keep the largest.

Brute force

Jumbled version

```
def A0(a):
for i in range(n):
for j in range(i,n):
for k in range(i,j+1):
largest = max(largest,s)
n,largest = len(a),0
return largest
s = 0
s = s+ a[k]
```

Brute force

```
1  def A0(a):
2      n,largest = len(a),0
3      for i in range(n):
4          for j in range(i,n):
5              s = 0
6              for k in range(i,j+1):
7                  s = s+ a[k]
8              largest = max(largest,s)
9      return largest
```

Testing

```
A0([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])
```

```
187
```

Correctness

- Big picture:
  - The outer two loops define every possible subarray $[i, j]$. For each of these, we find the sum and keep it if it exceed the previous largest sum. So, we clearly terminate with the largest of all.
- More details:
  - Entering the outer loop with `i` at value `k`, `largest` contains the largest sum of all sub-array `a[0..n-1]` to `a[k-1 .. n-1]`
  - Entering the inner loop with `j` at value `l`, `largest` contains the largest sum of all sub-array `a[0..n-1]` to `a[k-1 .. n-1]` and `a[k..0]` to `a[k..l-1]`. The inner loop adds one more sub array to compare and potentially update `largest`.
  - Since both loops go until the last position of the array, we will check all possible subarrays and keep the largest sum.

Runtime

- What should we count?

Runtime

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{i=n-1} \sum_{j=i}^{j=n-1} \sum_{k=i}^{k=j} 1 \\
&= \sum_{i=0}^{i=n-1} \sum_{j=i}^{j=n-1} (j - i + 1) \\
&= \sum_{i=0}^{i=n-1} \sum_{j=i}^{j=n-1} j + \sum_{j=i}^{j=n-1} (-i + 1) \\
&= \sum_{i=0}^{i=n-1} \frac{n^2 + n - i - 2in - 1}{2} \\
&= (n^2 + n)(n) - \frac{(n-1)n}{2}(-1 - 2n) - (n - 1) \\
&\in \Theta(n^3)
\end{aligned}
$$

Interlude

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

There are two approaches:

- Direct (requires a trick)
- By induction (boring but mechanical)

Can we do better than $\Theta(n^3)$?

- We can use `sum` to eliminate the inner loop. Does it change the runtime?

```
1  def A0(a):
2      n,largest = len(a),0
3      for i in range(n):
4          for j in range(i,n):
5              s = sum(a[i:j+1])
6              largest = max(largest,s)
7
8      return largest
```

- Discuss options.

Can we do better than $\Theta(n^3)$?

Hint: the sum from `a[i..j]` is related to the sum `a[i..j+1]`

Yes, we can!

Jumbled version.

```
def A1(a):
for i in range(n):
for j in range(i,n):
largest = max(largest,s)
n,largest = len(a),0
return largest
s += a[j]
s = 0
```

Yes, we can!

```
1  def A1(a):
2      n,largest = len(a),0
3      for i in range(n):
4          s = 0
5          for j in range(i,n):
6              s += a[j]
7              largest = max(largest,s)
8      return largest
```

Testing

```
A1([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])
```

```
187
```

Correctness

We terminate as the two loops are finite.
LI (outside loop): If we enter the loop with `i` at value `k`, then `largest` contains the largest sum of subarrays starting at positions smaller than `k`. Then we reset `s` and consider each subarray starting at `k`, keeping any one larger than our current best. So exiting the loop, we have considered all subarrays starting at positions smaller than or equal to `k`.
LI (inside loop): If we enter the loop with `j` at value `l`, then we have considered all subarrays starting at `k` and ending at all positions smaller than `l`. We then add `a[l]` to our current sum and keep it if it beats our current best. So, exiting the loop, we have considered all subarrays starting at `k` and ending at positions smaller than or equal to `l`.

Complexity

$$
\begin{aligned}
T(n) &= 2 + \sum_{i=0}^{i=n-1} \left(1 + \sum_{j=i}^{j=n-1} 2 \right) \\
&= 2 + \sum_{i=0}^{i=n-1} (1 + 2(n-i)) \\
&= 2 + n + 2n^2 - 2\frac{n(n-1)}{2} \\
&= 2 + n + n^2 - n \\
&\in \Theta(n^2)
\end{aligned}
$$

Can we do better than $\Theta(n^3)$?

Any other idea? Discuss.

How about pre-computing some information?

Keep an array of sums

$$a = [1, 3, -2, 7]$$
$$c = [0, 1, 4, 2, 9]$$

Why? Because the sum from i to j will be c[j+1]-c[i]

Pre-computing

```
1  def A2(a):
2      c = [0]*(len(a)+1)
3      for i in range(len(a)):
4          c[i+1] = c[i]+a[i]
5      largest = 0
6      for i in range (len(a)):
7          for j in range(i,len(a)):
8              s = c[j+1]-c[i]
9              largest = max(s,largest)
10     return largest
```

Testing

```
A2([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])
```

```
187
```

Complexity

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=0}^{i=n-1} 1 + 1 + \sum_{l=0}^{l=n-1} \sum_{u=l}^{u=n-1} 2 \\
&= 1 + \sum_{i=0}^{i=n-1} 1 + 1 + \sum_{l=0}^{l=n-1} 2(n-l) \\
&= 1 + \sum_{i=0}^{i=n-1} 1 + 1 + 2 \sum_{l=0}^{l=n-1} n - 2 \sum_{l=0}^{l=n-1} l \\
&= 1 + \sum_{i=0}^{i=n-1} 1 + 1 + 2n^2 - 2\frac{(n-1)n}{2} \\
&= 1 + n + 1 + 2n^2 - 2\frac{(n-1)n}{2} \\
&= 2 + n + 2n^2 - n^2 + n \\
&\in \Theta(n^2)
\end{aligned}
$$

Runtime

For an array of length n

- `A1` has runtime $n^2 + 2 \in \Theta(n^2)$
- `A2` has runtime $n^2 + 2n + 2 \in \Theta(n^2)$

Can we do better than $\Theta(n^2)$? Discuss

Think of design techniques we have seen at this point.

Let's try divide and conquer

- Split the array in two parts (Say A and B)
  - Find the maximum sum in each part separately (say M_A and M_B)
  - Pick the maximum of M_A and M_B
- Recursively (maybe?)

Any problem with this?

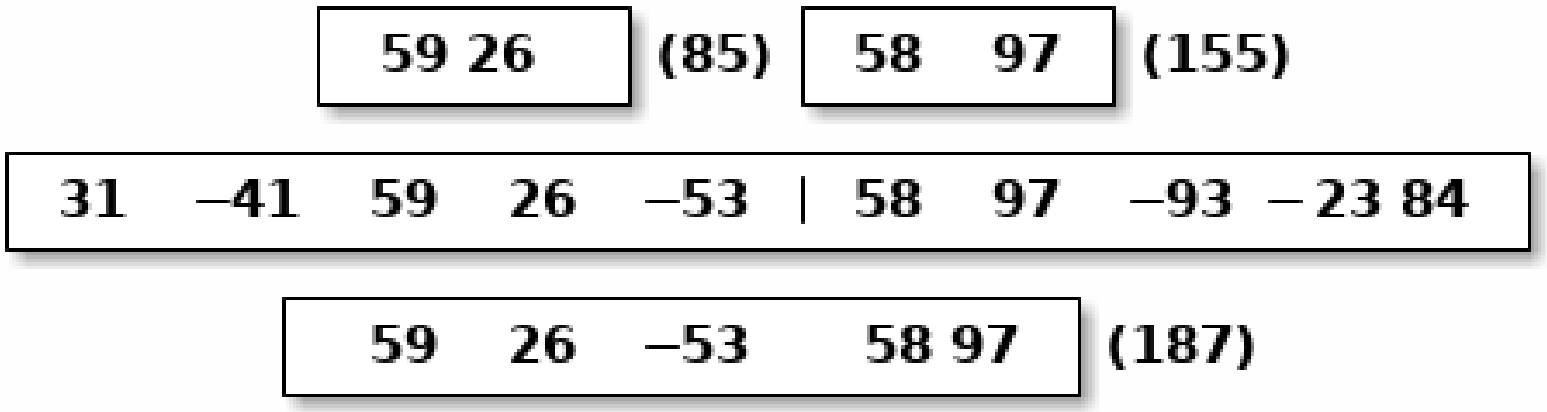
Divide and conquer

We split the array into two roughly equal parts.

- `Ma` is the maximum of the left part
- `Mb` is the maximum of the right part
- `Mc` is the maximum of the overlap

$$\max\{M_a, M_b, M_c\}$$

For example



Try to code this.

Divide-and-conquer

function A3:

- Split the array in two (equal) parts
- Recursively call A3 of left and right part
- Compute the max sub of the overlapping part
- Return the largest of the three choices.

Divide-and-conquer

Jumbled version

```
MA,MB,s,MCl = A3(a[0:m]),A3(a[m:n]),0,0
MCl = max(s,MCl)
MCr = max(MCr,s)
def A3(a):
for i in range(m+1,n):
for i in range(m,0,-1):
if n == 0:
if n == 1:
m = n//2
n = len(a)
return 0
return max(0,a[0])
return max(MA,MB,MCl+MCr)
s = s + a[i]
s = s + a[i]
s,MCr = 0,0
```

Divide-and-conquer

```
1  def A3(a):
2      n = len(a)
3      if n == 0:
4          return 0
5      if n == 1:
6          return max(0,a[0])
7      m = n//2
8      MA,MB,s,MCl = A3(a[0:m]),A3(a[m:n]),0,0
9      for i in range(m,0,-1):
10         s = s + a[i]
11         MCl = max(s,MCl)
12     s,MCr = 0,0
13     for i in range(m+1,n):
14         s = s + a[i]
15         MCr = max(MCr,s)
16     return max(MA,MB,MCl+MCr)
```

Testing

```
A3([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])
```

```
187
```

Complexity

What is the proper definition of the runtime function?

$$T(n) = \begin{cases} 1 & n \leq 1 \\ ? & \text{otherwise} \end{cases}$$

Complexity

The middle-part count could end-up reading the whole array!

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases}$$

Complexity

$$
\begin{aligned}
T(n) &= n + 2T(n/2) \\
&= n + 2[n/2 + 2T(n/4)] = 2n + 4T(n/4) \\
&= 2n + 4[n/4 + 2T(n/8)] = 3n + 8T(n/8) \\
&= \ldots \\
&= kn + 2^k T(n/(2^k)) \\
&= \ldots \\
&= n \log_2 n + Cn \\
&\in O(n \log_2 n)
\end{aligned}
$$

Runtime

Do you think A3 always beats A1 and A2? (Discuss)

Interlude: Three ways to solve recurrences

- Substitution (the method used up to now)
- Recursion tree (lots of drawing)
- Master theorem (memory required)

Use whichever way you want (but if you use MT, you must quote it).

Master theorem (to solve recurrences)

Consider the recurrence

$$T(n) = aT(n/b) + f(n)$$

- If $f(n) \in O(n^{\log_b a - \epsilon})$ then $T(n) \in \Theta(n^{log_b a})$
- If $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{log_b a} \log n)$
- If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) < (1-\epsilon)f(n)$ then $T(n) \in \Theta(f(n))$

Application of the MT

Consider

$$\mathrm{T(n)} = \begin{cases} 1 & n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases}$$

Does the MT apply? If so, identify a, b and $f(n)$.

$$T(n) = aT(n/b) + f(n)$$

Application of the MT

In which of the three cases are we? a=2, b=2, f(n) = n+2

- If $f(n) \in O(n^{\log_b a - \epsilon})$ then $T(n) = \Theta(n^{log_b a})$
- If $f(n) \in \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{log_b a} \log n)$
- If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) < (1-\epsilon)f(n)$ then $T(n) = \Theta(f(n))$

Application of the MT

Since $\log_2 2 = 1$, we are in the second case and $T(n) = n \log n$.

Can we do better than $\Theta(n \log n)$?

This is really two related questions?

- Is it computationally possible to do better?
- Can we find a way to do better?

Note:

- You could try to avoid reading the whole array in the 'middle' case.
- But that would be micro-optimization. No effect on the analysis.

Can we do better than $\Theta(n \log n)$?

Consider a sub array `a[0..i]` and assume we have the max subarray for it.
Now let us try to extend our knowledge to `a[0..i+1]`
The new maximum is either

- The same as for `a[0..i]` if `a[i+1] <= 0`.
- Some subarray ending at `i+1` if `a[i+1] > 0`.

Can you see how to use this?

Insight

Let us keep track of two values:

- The maximum subarray seen so far (`mf`)
- The maximum subarray ending at the current position (`mh`).
  Note that this `mh` is the empty subarray if ever the current sum goes negative.

Scanning algorithm

Can you code it?

Scanning algorithm

Jumbled version

```
def A4(a):
for i in range(n):
mf = max(mf,mh)
mf,mh,n = 0,0,len(a)
mh = max(mh+a[i], 0)
return mf
```

Scanning algorithm

```
1 def A4(a):
2       mf,mh,n = 0,0,len(a)
3       for i in range(n):
4           mh = max(mh+a[i], 0)
5           mf = max(mf,mh)
6       return mf
```

Testing

```
A4([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])
```

```
187
```

Understanding the algorithm

- Run it with paper and pen.
- Trace the code.

```
1  def A4(a):
2      mf,mh,n = 0,0,len(a)
3      print(" i a[i] mh  mf")
4      for i in range(n):
5          mh = max(mh+a[i], 0)
6          mf = max(mf,mh)
7          print("{0:2d} {1:3d} {2:3d} {3:3d}".format(i,a[i],mh,mf
8      return mf
```

Tracing the code

```
a=[10, -5, -7, 12, 3, -2, -4, 8, -25, 10, 2]
print(a)
A4(a)
```

```
[10, -5, -7, 12, 3, -2, -4, 8, -25, 10, 2]
 i a[i] mh  mf
 0  10  10  10
 1  -5   5  10
 2  -7   0  10
 3  12  12  12
 4   3  15  15
 5  -2  13  15
 6  -4   9  15
 7   8  17  17
 8 -25   0  17
 9  10  10  17
10   2  12  17
```

Correctness

The algorithms terminates as it is scanning each element exactly once.
Entering step where k, the variable `mh` contains the largest sum subarray ending at $k-1$, which could be zero, and variable `mf` contains the largest sum subarray anywhere in $[0, k-1]$.
This is trivially true at the start since both variables are zero.
Consider `i=k`. The algorithm adds `a[k]` to `mh` if this is still positive, indicating a sum ending at `k` or resets it to zero, indicating the empty subarray ending at `k`. Moreover it updates `mf` if this new `mh` is larger than any seen before. Therefore the loop invariant holds after the iteration.

Complexity

Trivially,

$$\Theta(n)$$

Can we possibly do better?

No, we need to read the array!

Formally

- We have an algorithm running in $\Theta(n)$
- We have a theoretical bound of $\Omega(n)$ since me must read an array of length n
- Therefore our algorithm is optimal! The best you can hope for is micro-optimization.

Morals (w.r.t. algorithm design)

- Recall that we went from $\Theta(n^3)$, to $\Theta(n^2)$, to $\Theta(n \log n)$ to $\Theta(n)$.
- Always be ready to produce the brute-force approach.
  - It gives you a baseline.
  - If it happens to be fast enough, you are done.
- Cleverer approaches:
  - Avoid computation by saving states.
  - Simplify computation by pre-computing.
  - Divide-and-conquer.
  - Scanning.
- Know how to
  - Mentally trace an algorithm.
  - Instrument code to display behaviour.

Scanning

- Decompose the problem as
  - Base case (empty array or array or length 1)
  - If I knew the answer to the question for array a[0,. . . ,i], can I extend this knowledge to array a[0,. . . ,i+1]?

This is a baby step towards a very powerful technique:

Dynamic Programming

(the decomposition can be more complex)

Generalization/Variations of the problem

- In A2, visit all sub-array of length 2, then of length 3, etc. . .
- What if you were asked to return the subarray instead of the sum?
- What if, instead of an 1-D array, we have a 2-D array, a k-D array?
- Consider a potential A5:
  - Find the subarray starting at index 0 ending at `e` with the largest sum.
  - Start scanning backward from `e` down to `s` the subarray ending at `e` of maximum sum.
  - Return the sum of the subarray `s` to `e`.
    - Is that algorithm correct?
    - What is its runtime?
- What if we did not assume zero for the empty array? What would change?

Recall

```
def A4(a):
    mf,mh,n = 0,0,len(a)
    for i in range(n):
        mh = max(mh+a[i], 0)
        mf = max(mf,mh)

    return mf
```

Modify to return the starting and ending index of the subarray.

Returning the indices

```
def A4ix(a):
    mf,mh,n,s,e,sh,eh = 0,0,len(a),0,-1,0,-1
    for i in range(n):
        if mh+a[i] > 0:
            eh,mh = i,mh+a[i]
        else:
            mh,sh = 0,i+1
        if mh > mf:
            mf,s,e = mh,sh,eh
    return mf,s,e
from random import randint
def testa4ix():
    for a,m0,s0,e0 in [([],0,0,-1),
                       ([1],1,0,0),
                       ([1,2],3,0,1),
                       ([4,-2],4,0,0),
                       ([4,-2,3],5,0,2),
                       ([4,-3,-2,5,2],7,3,4)]:
        (m,s,e) = A4ix(a)
```

Coffee can problem

Problem

You are given a can of coffee beans. Some are white, others black. At each step, take two random beans out. If they are the same colour, throw them out and add a black bean to the can. Otherwise, return the white to the can and discard the black.

Questions

- Does the process terminate?
- If so, what is the colour of the last bean?

Class Project

Red, White and Blue

The problem is to sort an array containing exactly three symbols, say red, white and blue so that all reds come together, followed by all whites, followed finally by all blues.
Alternatively you can use 0, 1 and 2. The problem is the same.
Your objective is to do this in $O(n)$ time without additional memory.
Therefore none of the general-purpose sorting algorithms can be used and neither can you create a new array. You must do this in-place.