Max sum sub-array problem

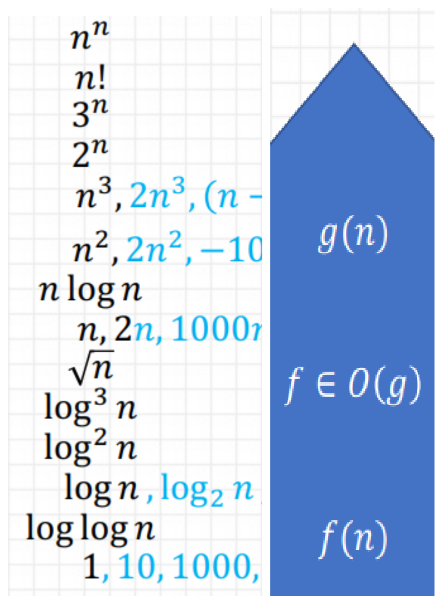
- D&C (time=O(nlogn), space=O(logn)
  - Cut input in half
  - If left bound == right bound, return max(0, arr[left])
  - find max sum of left
  - find max sum of right

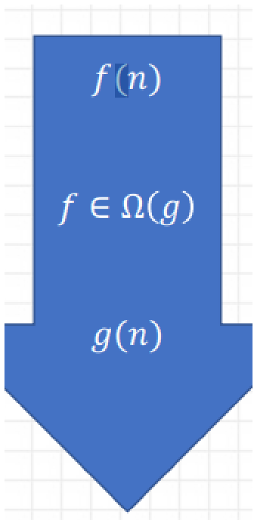
```
def maxCross(array, low, mid, high):
    sum_left = -10000000000
    sum_right = -10000000000
    sumVar = 0
    for i in range(mid, low, -1):
        sumVar += array[i]
        if sumVar > sum_left:
            sum_left = sumVar

    sumVar = 0
    for i in range(mid + 1, high, 1):
        sumVar += array[i]
        if sumVar > sum_right:
            sum_right = sumVar

    return sum_left + sum_right
```

  - Do ^^ for finding the max of things starting in one half and ending in the other.
  - Return max(left, right, maxCross)
- DP (time=O(n), space=O(1)
  - Set a current sum to -inf
  - Set max sum to -inf
  - For loop through array
    - Currsum = max(currsum + arr[i], arr[i])
    - Maxsum = max(maxSum, currSum)
  - Return maxSum

Master Theorem

- $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$
  - A is number of subproblems we make
  - B is factor by which subproblem size decreases
  - $K = \log_b n$
  - f(n) is difficulty to divide and recombine subproblems
- Geometric series reminders
  - R =/= 1
    - $\frac{1-r^k}{1-r}$
  - R = 1
    - K
  - R < 1
    - $\frac{1}{1-r}$
  - R is $\frac{a}{b^d}$ where d is the exponent of f(n)
- So to solve T(n), we take the result from the geometric series stuff, use K+1 instead of K, and multiply by f(n).
- $$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \quad (\text{case } 1) \\ \Theta\left(n^d \log n\right), & \text{if } a = b^d \quad (\text{case } 2) \\ \Theta\left(n^d\right), & \text{if } a < b^d \quad (\text{case } 3) \end{cases}$$
- Limitations: Master Theor no work
  - If T(n) is not monotone
    - T(n) = sin(n)
  - If f(n) is not polynomial
    - f(n) = 2^n
  - If b cannot be expressed as a constant





DP

- Show optimal substructure
  - Base case and recurrence relation
- Show subproblems overlap
  - # of distinct subproblems is polynomial
- Algo must construct optimal solutions for subproblems once and reuse stored results

Big-O Properties

- Constants don't affect
- $f_1 * f_2 is\ O(g_1 g_2)$
- $f_1 + f_2\ is\ O(max(g_1, g_2))$
- If f is O(g), and g is O(h), then f is O(h)

Log properties

- $log_b n = log_b a * log_a n$

Mergesort

- Break down array in halves until individual elements remain
- Add elements to temp array in order of magnitude, recombine array into one

Array Sorting Algorithms

Algorithm	Running time			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Quicksort

```
def quicksort(arr):
  def partition(l, r):
    pivot = r-1
    index = l
    print("pivot value")
    print(arr[pivot])
    print("pivot index")
    print(pivot)
    for i in range(l, pivot):
      if arr[i] < arr[pivot]:
        arr[i], arr[index] = arr[index],
arr[i]
        print("in for loop ")
        print(arr)
        index += 1
    arr[pivot], arr[index] = arr[index],
arr[pivot]
    print('after for loop')
    print(arr)
    return index

  def sort(l, r):
    if l < r:
      p = partition(l, r)
      # conquer
      sort(l, p)
      sort(p+1, r)

  sort(0, len(arr))
```