Week 3

Maximum Subarray Sum

Thorben Klabunde

www.th-kl.ch

06.10.2025

Agenda

1. Mini-Quiz
2. Assignment
3. Theory Recap
4. Additional Practice
5. Peer Grading

Mini-Quiz

Assignment

Ex. 2.2

Exercise 2.2 *Fibonacci numbers* **(1 point)**.

There are a lot of neat properties of the Fibonacci numbers that can be proved by induction. Recall that the Fibonacci numbers are defined by $f_0 = 0, f_1 = 1$ and the recursion relation $f_{n+1} = f_n + f_{n-1}$ for all $n \geq 1$. For example, $f_2 = 1, f_5 = 5, f_{10} = 55, f_{15} = 610$.

Prove that $f_n \geq \frac{1}{3} \cdot 1.5^n$ for $n \geq 1$.

In your solution, you should address the base case, the induction hypothesis and the induction step.

Ex. 2.2 - ctd.

Ex. 2.4a)

Exercise 2.4 *Asymptotic growth of* $\sum_{i=1}^{n} \frac{1}{i}$ **(1 point).**

The goal of this exercise is to show that the sum $\sum_{i=1}^{n} \frac{1}{i}$ behaves, up to constant factors, as $\log(n)$ when n is large. Formally, we will show $\sum_{i=1}^{n} \frac{1}{i} \leq O(\log n)$ and $\log n \leq O(\sum_{i=1}^{n} \frac{1}{i})$ as functions from $\mathbb{N}_{\geq 2}$ to $\mathbb{R}^+$.

For parts (a) to (c) we assume that $n = 2^k$ is a power of 2 for $k \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. We will generalise the result to arbitrary $n \in \mathbb{N}$ in part (d). For $j \in \mathbb{N}$, define

$$S_j = \sum_{i=2^{j-1}+1}^{2^j} \frac{1}{i}.$$

(a) For any $j \in \mathbb{N}$, prove that $S_j \leq 1$.

Hint: *Find a common upper bound for all terms in the sum and count the number of terms.*

Ex. 2.4 - ctd.

Ex. 2.4b)

(b) For any $j \in \mathbb{N}$, prove that $S_j \geq \frac{1}{2}$.

Ex. 2.4c)

(c) For any $k \in \mathbb{N}_0$, prove the following two inequalities

$$\sum_{i=1}^{2^k} \frac{1}{i} \leq k+1$$

and

$$\sum_{i=1}^{2^k} \frac{1}{i} \geq \frac{k+1}{2}.$$

Hint: *You can use that* $\sum_{i=1}^{2^k} \frac{1}{i} = 1 + \sum_{j=1}^{k} S_j$. *Use this, together with parts (a) and (b), to prove the required inequalities.*

Ex. 2.4d)

(d)* For arbitrary $n \in \mathbb{N}$, prove that

$$\sum_{i=1}^{n} \frac{1}{i} \leq \log_2(n) + 2$$

and

$$\sum_{i=1}^{n} \frac{1}{i} \geq \frac{\log_2 n}{2}.$$

Hint: *Use the result from part (c) for* $k_1 = \lceil \log_2 n \rceil$ *and* $k_2 = \lfloor \log_2 n \rfloor$. *Here, for any* $x \in \mathbb{R}$, $\lceil x \rceil$ *is the smallest integer that is at least* x *and* $\lfloor x \rfloor$ *is the largest integer that is at most* x. *For example,* $\lceil 1.5 \rceil = 2$, $\lfloor 1.5 \rfloor = 1$ *and* $\lceil 3 \rceil = \lfloor 3 \rfloor = 3$. *In particular, for any* $x \in \mathbb{R}$, $x \leq \lceil x \rceil < x + 1$ *and* $x \geq \lfloor x \rfloor > x - 1$.

Ex. 2.5a)

Exercise 2.5 *Asymptotic growth of* $\ln(n!)$.

Recall that the factorial of a positive integer n is defined as $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$. For the following functions n ranges over $\mathbb{N}_{\geq 2}$.

(a) Show that $\ln(n!) \leq O(n \ln n)$.

Hint: *You can use the fact that* $n! \leq n^n$ *for* $n \in \mathbb{N}_{\geq 2}$ *without proof.*

Ex. 2.5b)

(b) Show that $n \ln n \leq O(\ln(n!))$.

Hint: *You can use the fact that* $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n!$ *for* $n \in \mathbb{N}_{\geq 2}$ *without proof.*

Theory Recap

The Maximum Subarray Sum Problem

Problem Definition

Given an array of integers arr, find the **contiguous** subarray (incl. $\varnothing$) which has the **largest sum**.

Consider the array:

$\text{arr} = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$

with max. subarray sum 6, corresponding to

$[4, -1, 2, 1]$.

```
public class MaxSubarraySum {
    int[] arr;
    int n;

    MaxSubarraySum(int[] arr) {
        this.arr = arr;
        n = arr.length;
    }

    ...
}
```

We use this problem (it has practical applications as well!) to illustrate and analyze different algorithm design paradigms: brute force, divide-and-conquer, and dynamic programming.

Approach 1: Naive Cubic Time $O(N^3)$

Idea: Iterate through all possible start indices i, all possible end indices j, and for each pair (i, j), sum up all elements from i to j.

```
Result naiveSumCubic() {
    int startIdx = -1;
    int endIdx = -1;
    int maxSum = 0;
    for (int i=0; i<n; i++) {
        for (int j=i; j<n; j++) {
            int sum = 0;
            for (int k=i; k<=j; k++) {
                sum += arr[k];
            }
            if (sum > maxSum) {
                startIdx = i;
                endIdx = j;
                maxSum = sum;
            }
        }
    }
    return new Result(maxSum, startIdx, endIdx);
}
```

How do we analyze the runtime of this program?

Break the structure down to the essentials!

$$C_1 + C_2 \cdot \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1, \quad \text{for some } C_1, C_2 \in \mathbb{N}$$

Approach 2: Improved Quadratic Time $O(N^2)$

Improvement: The third loop is inefficient. We can avoid it by reusing the sum from the previous step as we extend the subarray to the right.

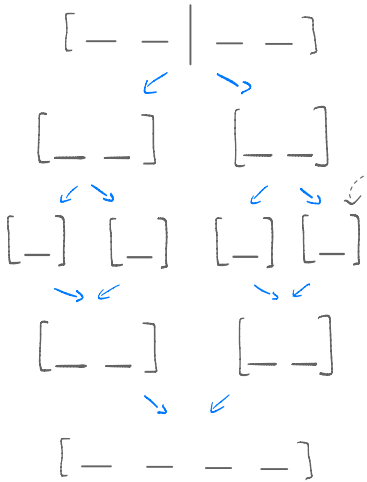
```
Result naiveSumSquare() {
    int startIdx = -1;
    int endIdx = -1;
    int maxSum = 0;
    for (int i=0; i<n; i++) {
        int sum = 0;
        for (int j=i; j<n; j++) {
            sum += arr[j]; // Just add the next element
            if (sum > maxSum) {
                startIdx = i;
                endIdx = j;
                maxSum = sum;
            }
        }
    }
    return new Result(maxSum, startIdx, endIdx);
}
```

Runtime:

We have eliminated one loop and obtain:

$$C_1 + C_2 \cdot \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1$$

Approach 3a: Standard Divide and Conquer $O(N \log N)$



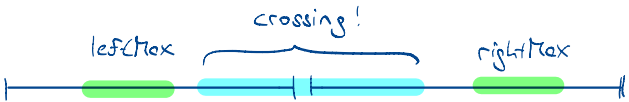
DIVIDE

Divide into smaller, more manageable problems until reaching base case with trivial solution

CONQUER

Assemble solution to larger problem from smaller problems.

Note: We have to be careful not to miss solutions!



Approach 3: Standard Divide and Conquer $O(N \log N)$

Idea: The max subarray is either (1) in the left half, (2) in the right half, or (3) it crosses the midpoint. The crossing sum is found by scanning outwards from the middle.

```
int dcNLogN(int[] arr, int left, int right) {
    // Base Case: Only one element
    if (left == right) return Math.max(0, arr[left]);

    // 1. Divide
    int mid = left + (right - left) / 2;

    // 2. Conquer
    int leftMax = dcNLogN(arr, left, mid);
    int rightMax = dcNLogN(arr, mid + 1, right);

    // 3. Combine: Find max crossing the midpoint
    int leftCrossingMax = Integer.MIN_VALUE;
    int currentSum = 0;
    for (int i = mid; i >= left; i--) {
        currentSum += arr[i];
        if (currentSum > leftCrossingMax) {
            leftCrossingMax = currentSum;
        }
    }
    int rightCrossingMax = Integer.MIN_VALUE;
    currentSum = 0;
    for (int i = mid + 1; i <= right; i++) {
        currentSum += arr[i];
        if (currentSum > rightCrossingMax) {
            rightCrossingMax = currentSum;
        }
    }
    int crossingMax = leftCrossingMax + rightCrossingMax;
    int result = Math.max(Math.max(leftMax, rightMax), crossingMax);
    return Math.max(0, result);
}
```

How do we analyze recursive algorithms?

We formulate the recurrence relation:

Approach 4: Linear DP $O(N)$

Idea: Break the problem down into subproblems that we can easily build on!

```
Result linearPassSum() {
    // DP[i] = max subarray sum ending at arr[i-1]
    int[] DP = new int[n+1];
    int maxSum = 0; int startIdx = endIdx = -1;
    int currentStartIdx = -1;
    DP[0] = 0;

    for (int i=1; i<n+1; i++) {
        int prevSum = DP[i-1];
        int currentVal = arr[i-1];

        if (prevSum + currentVal > 0) {
            DP[i] = prevSum + currentVal;
            // start index remains the same
        } else {
            DP[i] = 0; // Start a new empty subarray
            currentStartIdx = i;
        }

        if (DP[i] > maxSum) {
            maxSum = DP[i];
            endIdx = i-1;
            startIdx = currentStartIdx;
        }
    }
    return new Result(maxSum, startIdx, endIdx);
}
```

Correctness:

Meaning of each entry (DP[i]):

Recursion:

Correctness:

Approach 4: Linear DP $O(N)$

Idea: Break the problem down into subproblems that we can easily build on!

```
Result linearPassSum() {
    // DP[i] = max subarray sum ending at arr[i-1]
    int[] DP = new int[n+1];
    int maxSum = 0; int startIdx = endIdx = -1;
    int currentStartIdx = -1;
    DP[0] = 0;

    for (int i=1; i<n+1; i++) {
        int prevSum = DP[i-1];
        int currentVal = arr[i-1];

        if (prevSum + currentVal > 0) {
            DP[i] = prevSum + currentVal;
            // start index remains the same
        } else {
            DP[i] = 0; // Start a new empty subarray
            currentStartIdx = i;
        }

        if (DP[i] > maxSum) {
            maxSum = DP[i];
            endIdx = i-1;
            startIdx = currentStartIdx;
        }
    }
    return new Result(maxSum, startIdx, endIdx);
}
```

Runtime:

Divide-and-Conquer - Geht es besser?

Recall: The max sum is the maximum of: (1) max sum in left half, (2) max sum in right half, and (3) max sum crossing the midpoint.

```
DCResult divideAndConquerSum(int left, int right) {
    if (right == left) { // Base Case
        int val = Math.max(0, arr[left]);
        int idx = (arr[left] >= 0) ? left : -1;
        return new DCResult(val, idx, val, idx, val, idx, idx, arr[left]);
    }
    int mid = left+((right-left)/2);
    DCResult l = divideAndConquerSum(left, mid);
    DCResult r = divideAndConquerSum(mid+1, right);

    // Combine step (O(1) work)
    int leftMax = Math.max(l.leftMax, l.total + r.leftMax);
    int rightMax = Math.max(r.rightMax, r.total + l.rightMax);
    int maxSum;
    if (l.rightMax + r.leftMax > Math.max(l.maxSum, r.maxSum)) {
        maxSum = l.rightMax + r.leftMax;
    } else {
        maxSum = Math.max(l.maxSum, r.maxSum);
    }
    int total = l.total + r.total;
    // (index tracking logic omitted for brevity)
    return new DCResult(leftMax, ..., rightMax, ..., maxSum, ..., total);
}
```

Runtime:

Divide-and-Conquer - Geht es besser?

Recall: The max sum is the maximum of: (1) max sum in left half, (2) max sum in right half, and (3) max sum crossing the midpoint.

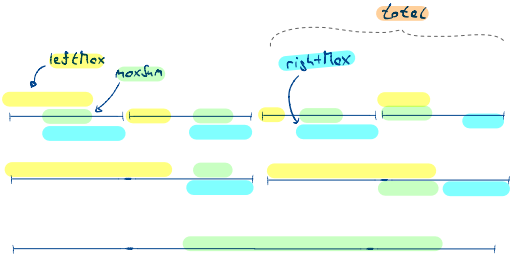
```
DCResult divideAndConquerSum(int left, int right) {
    if (right == left) { // Base Case
        int val = Math.max(0, arr[left]);
        int idx = (arr[left] >= 0) ? left : -1;
        return new DCResult(val, idx, val, idx, val, idx, idx, arr[left]);
    }
    int mid = left+((right-left)/2);
    DCResult l = divideAndConquerSum(left, mid);
    DCResult r = divideAndConquerSum(mid+1, right);

    // Combine step (O(1) work)
    int leftMax = Math.max(l.leftMax, l.total + r.leftMax);
    int rightMax = Math.max(r.rightMax, r.total + l.rightMax);
    int maxSum;
    if (l.rightMax + r.leftMax > Math.max(l.maxSum, r.maxSum)) {
        maxSum = l.rightMax + r.leftMax;
    } else {
        maxSum = Math.max(l.maxSum, r.maxSum);
    }
    int total = l.total + r.total;
    // (index tracking logic omitted for brevity)
    return new DCResult(leftMax, ..., rightMax, ..., maxSum, ..., total);
}
```

Runtime:

All the required information is there.

No need to recompute!



Visualizing Runtime of Recursive DnC

Questions?

Bounding Sums with Integrals

Recall From Last Week

We showed that for any constant $k \geq 1$, the sum of the first n k-th powers is in $\Theta(n^{k+1})$.

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

We proved this by finding constants c_1 and c_2 to establish the upper and lower bounds directly from the sum's properties.

A New Technique: Using Integrals

The core idea: The sum $\sum_{i=1}^{n} f(i)$ can be seen as the total area of n rectangles, each with width 1 and height $f(i)$. This area can be bounded by the area under the curve of the continuous function $f(x)$.

Bounding Sums with Integrals

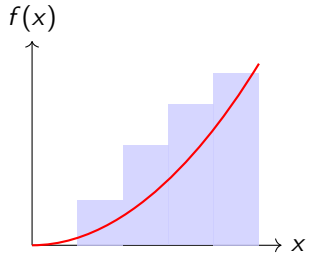
For any **monotonically increasing** function $f(x)$:

Lower Bound: The sum is at least the area under the curve from 0 to n.

$$\sum_{i=1}^{n} f(i) \geq \int_{0}^{n} f(x)\,dx$$

Upper Bound: The sum is at most the area under the curve from 1 to $n+1$.

$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)\,dx$$



Area under curve bounds sum

Bounding Sums with Integrals

Example: Applying this to $\sum_{i=1}^{n} i^k$

Let $f(x) = x^k$. This function is monotonically increasing for $x \geq 0$.

Lower Bound (Ω):

$$\sum_{i=1}^{n} i^k \geq \int_0^n x^k \, dx = \left[\frac{x^{k+1}}{k+1}\right]_0^n = \frac{n^{k+1}}{k+1} \implies \Omega(n^{k+1})$$

Upper Bound (O):

$$\sum_{i=1}^{n} i^k \leq \int_1^{n+1} x^k \, dx = \left[\frac{x^{k+1}}{k+1}\right]_1^{n+1} = \frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1} \implies O(n^{k+1})$$

Since the sum is both $\Omega(n^{k+1})$ and $O(n^{k+1})$, we conclude it is $\Theta(n^{k+1})$.

Note on Monotonically Decreasing Functions

What if the function is decreasing?

The same technique works, but the inequalities for the bounds are flipped. For a monotonically **decreasing** function $f(x)$:

$$\int_1^{n+1} f(x)\,dx \leq \sum_{i=1}^{n} f(i) \leq f(1) + \int_1^n f(x)\,dx$$

Note: The upper bound is the first term $f(1)$ plus the integral over the rest of the terms.

Your Turn!

Claim: The Harmonic Series $H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$.

Additional Practice

Counting Loop Iterations

/ 4 P

a) *Counting loop iterations:* For the following code snippets, derive an expression for the number of times f is called. Simplify the expression as much as possible and state it in Θ-notation.

i) Snippet 1:

Algorithm 1

```
for j = 1, ..., n do
    for k = j^2, ..., (j + 1)^2 do
        f()
```

How fast does $n!$ grow?

Show that $n! \approx (\frac{n}{e})^n$.

Peer Grading

Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 2.4**

Each group grades the group below in the table I sent you (resp. the last one grades the first one). Please send the other group your solution. If you don't get their solution, please contact me so I can send it to you.