ALGORITHMS & DATA STRUCTURES

11TH OCTOBER 2021

TODAY’S PROGRAM

- Recap: Maximum Subarray Sum
- Bonus Exercises
- Induction Exercise

MAXIMUM SUBARRAY SUM

THE PROBLEM

- **Given:** Sequence of integers a_1, ..., a_n (each can be positive or negative)

- **Goal:** compute indices i, j such that the sum a_i, ..., a_j is maximized

7, 10, -100, 25, -6, 1

7, 10, -100, 25, -6, 1

O(n^3) ALGORITHM

Try all possible indices i, j

Compute the sum

Keep the maximum

Why O(n^3)? O(n) choices for i, O(n) choices for j, O(n) time to compute each sum.

O(n^2) algorithm

We still try all possible i, j

But we save time in computing the sum

<u>Preprocessing step:</u> compute prefix sum array in O(n)

We can compute the sum for indices i, j in constant time (instead of linear time)

DIVIDE AND CONQUER: O(n log n) algorithm

$\text{ALGO}(a, i, j)$

if $(i = j)$ return a_i

left $\leftarrow \text{ALGO}(a, i, \frac{i+j}{2})$

right $\leftarrow \text{ALGO}(a, \frac{i+j}{2}, j)$

mid $\leftarrow$ ...

return max(left, right, mid)

<u>mid:</u> precompute prefix / suffix sum

mid = max suffix to the left + prefix to the right

$\Rightarrow O(n)$

$$T(n) \le 2T\left(\frac{n}{2}\right) + O(n)$$
$$\le 2\left(2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\right) + O(n)$$
$$\le 4T\left(\frac{n}{4}\right) + 2\,O(n)$$
$$\le \dots \longrightarrow \text{INDUCTION!}$$
$$\le k\,T\left(\frac{n}{k}\right) + \log_2 k \cdot O(n)$$
$$k = n \Rightarrow n\,T(1) + \log_2 n \cdot O(n)$$
$$\le O(n \log n)$$

TRICK: O(n) ALGORITHM

```
local_max = 0
global_max = 0
for i = 1 ... n
    local_max = local_max + a_i
    if local_max < 0
        local_max = 0
    global_max = max( global_max,
                      local_max)
```

EXERCISE 2.2

$(n^2 - n + 1)^2 \leq O(n^4)$ and $n^4 \leq O((n^2 - n + 1)^2)$.

$$\lim_{n \to \infty} \frac{(n^2 - n + 1)^2}{n^4} = \lim_{n \to \infty} \frac{n^4 - 2n^3 + 3n^2 - 2n + 1}{n^4}$$

$$= 1$$

$\Longrightarrow$ BOTH ARE TRUE

$$\sqrt{n} \leq O(\sqrt[3]{n \log n}).$$

$$\lim_{n \to \infty} \frac{n^{1/2}}{(n \log n)^{1/3}} = \lim_{n \to \infty} \frac{n^{1/6}}{(\log n)^{1/3}} = \infty$$

$$\Rightarrow \sqrt{n} \not\leq O(\sqrt[3]{n \log n})$$

$$\log_{100}^2(n) \leq O(\log_2(n^{100})).$$

$$\log_{100}(n) = \frac{\log_2 n}{\log_2 100}$$

$$\lim_{n \to \infty} \frac{\left(\log_2 n / \log_2 100\right)^2}{100 \log_2 n} = \lim_{n \to \infty} \frac{\log_2 n}{100 \cdot (\log_2 100)^2} = \infty$$

$$\Rightarrow \log_{100}^2(n) \notin O(\log_2(n^{100}))$$

$$\sum_{k=1}^{n} \left(k^2 e^k + \ln^3 k\right) \le O(3^n).$$

$$\sum_{k=1}^{n} (k^2 e^k + \ln^3 k) \le \sum_{k=1}^{n} (n^2 e^n + \ln^3 n)$$

$$= n^3 e^n + n \ln^3 n$$

$$\lim_{n \to \infty} \frac{n^3 e^n + n \ln^3 n}{3^n} = 0 \qquad (e < 3)$$

$\Rightarrow$ TRUE

$$\sum_{k=0}^{n} 2^k \leq O(2^n).$$

Claim: $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1 \quad (*)$

n = 0 $\quad 2^0 = 1$

$2^{0+1} - 1 = 1 \quad \checkmark$

IH $\quad (*)$ holds for an n

IS $\quad \sum_{k=0}^{n+1} 2^k = \sum_{k=0}^{n} 2^k + 2^{n+1}$

$\overset{IH}{=} 2^{n+1} - 1 + 2^{n+1}$

$= 2^{n+2} - 1$

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$$

$$= 2 \cdot 2^n - 1$$

$$\leq 2 \cdot 2^n$$

$$\leq O(2^n)$$

EXERCISE 2.4

Many algorithms are iterative in the sense that they repeat n iterations of some procedure. The number of iterations n is usually monotonically related to the size of the input, i.e., bigger inputs require more

iterations. Furthermore, it is often the case that later iterations take more time. More precisely, if $t(k)$ is the time taken by the k-th iteration, then $t(i) \leq t(j)$ for all $i \leq j$. Clearly, the total running time $T(n)$ of such an algorithm satisfies

$$T(n) = \sum_{k=1}^{n} t(k).$$

In this exercise, we are interested in analyzing the asymptotic growth of $T(n)$ in terms of that of $t(n)$. As we previously mentioned, in this exercise we always assume that the function $t : \mathbb{N} \to \mathbb{N}$ is nondecreasing.

a) Show that for arbitrary nondecreasing function $t : \mathbb{N} \to \mathbb{N}$, we always have $T(n) \leq O(n \cdot t(n))$.

$$T(n) = \sum_{k=1}^{n} t(k)$$

t nondecreasing

$$\leq \sum_{k=1}^{n} t(n)$$

$$= n \cdot t(n)$$

b) Assume that $t(n)$ grows polynomially, i.e., there exist real numbers $\beta > 0$ and $C \geq 1$ such that for all $n \in \mathbb{N}$, $\frac{1}{C} \cdot n^\beta \leq t(n) \leq C \cdot n^\beta$. Show that $n \cdot t(n) \leq O(T(n))$.

Hint: *Use the fact that for every* $n \geq 2$*, we have* $\sum_{k=\lceil \frac{n}{2} \rceil}^{n} k^\beta \geq \frac{n}{2} \cdot \left(\frac{n}{2}\right)^\beta$*, where* $\lceil x \rceil$ *denotes the smallest integer* ℓ *satisfying* $\ell \geq x$*.*

$$n \cdot t(n) \leq n \cdot C \cdot n^\beta = \frac{n}{2} \cdot \left(\frac{n}{2}\right)^\beta \cdot 2^{\beta+1} \cdot C$$

$$\leq C \cdot 2^{\beta+1} \cdot \sum_{k=\lceil \frac{n}{2} \rceil}^{n} k^\beta$$

$$\leq C \cdot 2^{\beta+1} \cdot \sum_{k=\lceil \frac{n}{2} \rceil}^{n} C \cdot t(k)$$

$$= \underbrace{C^2 \cdot 2^{\beta+1}}_{:= \tilde{C}} \cdot \sum_{k=1}^{n} t(k)$$

$$= \tilde{C} \cdot T(n) \leq O(T(n))$$

c) Show that for arbitrary nondecreasing function $t : \mathbb{N} \to \mathbb{N}$, we always have $t(n) \leq O(T(n))$.

$$t(n) \leq \sum_{u=1}^{n} t(u) = T(n)$$

d) Assume that $t(n)$ grows exponentially, i.e., there exist real numbers $\alpha > 1$ and $C \geq 1$ such that for all $n \in \mathbb{N}$, $\frac{1}{C} \cdot \alpha^n \leq t(n) \leq C \cdot \alpha^n$. Show that $T(n) \leq O(t(n))$.

$$T(n) = \sum_{k=1}^{n} t(k)$$

$$\leq \sum_{k=1}^{n} C \cdot \alpha^k$$

Ex. 1.2

$$\leq C \cdot \frac{\alpha^{n+1} - 1}{\alpha - 1}$$

$$\leq C \cdot \frac{\alpha^{n+1}}{\alpha - 1}$$

$$= \frac{C}{\alpha - 1} \cdot \alpha \cdot \alpha^n$$

$$= \tilde{C} \cdot \alpha^n$$

$$\leq \tilde{C} \cdot C \cdot t(n)$$

$$\leq O(t(n))$$

EXERCISE 2.5

The simplest algorithm that we can think of is the following procedure:

- Check whether $d = m$ is a common divisor for n and m. If yes, output d. Otherwise, go to the next step.
- Check whether $d = m - 1$ is a common divisor for n and m. If yes, output d. Otherwise, go to the next step.
- Then, we similarly check $m - 2, m - 3, \ldots, 1$.

Let $T_{\text{simple}}(n, m)$ be the number of iterations (steps) that are taken by the above simple algorithm to find the greatest common divisor.

a) For fixed $n \geq 1$, determine $\min_{1 \leq m \leq n} T_{\text{simple}}(n, m)$ and $\max_{1 \leq m \leq n} T_{\text{simple}}(n, m)$?

$$\min_{1 \leq m \leq n} T(n,m) = T(n,n) = 1$$

$$\max_{1 \leq m \leq n} T(n,m) = T(n,n-1) = n-1$$

$$\gcd(n,n) = n$$

$$\gcd(n,n-1) = 1$$

- Define $n_0 = n$ and $n_1 = m$.
- For every $i \geq 1$, as long as, $n_i > 0$, we perform the division with remainder of n_{i-1} by n_i, i.e., we find the unique q_i, r_i that satisfy $n_{i-1} = q_i n_i + r_i$ and $0 \leq r_i < n_i$, and then we define $n_{i+1} = r_i$.
- When we reach i for which $n_i = 0$, we stop and output n_{i-1}.

This is called the Euclidean algorithm for computing the greatest common divisor.

e) Show that for every $i \geq 2$, we have $n_i < \frac{n_{i-2}}{2}$.

$n_i < n_{i-1}$

$\boxed{\text{Case 1} \quad n_{i-1} \leq \frac{n_{i-2}}{2}}$ $n_i < n_{i-1} \leq \frac{n_{i-2}}{2}$ ✓.

$\boxed{\text{Case 2} \quad n_{i-1} > \frac{n_{i-2}}{2}}$ $n_{i-2} < 2n_{i-1} \Rightarrow n_{i-2} - n_{i-1} < n_{i-1}$

$n_{i-2} = n_{i-1} + (n_{i-2} - n_{i-1})$

$\Rightarrow q_{i-1} = 1, \; r_{i-1} = n_{i-2} - n_{i-1}$

$n_i = r_{i-1} = n_{i-2} - n_{i-1} < \frac{n_{i-2}}{2}$ ✓.

Let $T_E(n,m)$ be the number of iterations that are taken by the Euclidean algorithm to find the greatest common divisor of n and m (where $n \geq m$).

f) Show that $T_E(n,m) \leq O(\log(n))$.

e) $n_i < \frac{n_{i-2}}{2}$

$n_2 < \frac{n_0}{2}$

$n_4 < \frac{n_2}{2} < \frac{n_0}{4}$

... Claim $n_{2k} < \frac{n_0}{2^k}$

$\boxed{k=1}$ $n_2 < \frac{n_0}{2}$ ✓

IH Claim holds for a k

IS $n_{2(k+1)} < \frac{n_{2k}}{2} \overset{IH}{<} \frac{n_0}{2^{k+1}}$ $\square$

$n_{2k} < \frac{n_0}{2^k}$

Pick $k = \log_2 n_0$

$\Rightarrow n_{2k} < \frac{n_0}{n_0} = 1$

If n_{2k} is not defined, the algo needs less then $\log_2 n_0$ steps. If it is defined, it is 0 and so we are done

$\Rightarrow T_E(n,m) \leq 2\log_2 n_0$

INDUCTION

2. Let x be a real number. Prove via mathematical induction that for every positive integer n, we have

$$(1+x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i ,$$

where

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} .$$

We use a standard convention $0! = 1$, so $\binom{n}{0} = \binom{n}{n} = 1$ for every positive integer n.

Hint: You can use the following fact without justification: for every $1 \leq i \leq n$,

$$\binom{n}{i} + \binom{n}{i-1} = \binom{n+1}{i} .$$

$\boxed{n=1}$ $(1+x)^1 = (1+x)$

$$\sum_{i=0}^{1} \binom{1}{i} x^i = \binom{1}{0} x^0 + \binom{1}{1} x = 1+x$$

$\boxed{IH}$ $(*)$ holds for an n

$\boxed{IS}$ $(1+x)^{n+1} = (1+x)\cdot(1+x)^n \overset{IH}{=} (1+x) \sum_{i=0}^{n} \binom{n}{i} x^i$

$$= \sum_{i=0}^{n} \binom{n}{i} x^i + \sum_{i=1}^{n+1} \binom{n}{i-1} x^i$$

$$= \binom{n}{0} x^0 + \sum_{i=1}^{n} \left[\binom{n}{i} + \binom{n}{i-1}\right] x^i + \binom{n}{n} x^{n+1}$$

$$= 1 + \sum_{i=1}^{n} \binom{n+1}{i} x^i + x^{n+1} = \sum_{i=0}^{n+1} \binom{n+1}{i} x^i$$