Office hours posted

Pset 1 is out! Due: 2/18, 11:59 PM

Two parts: (A) Theory (B) Programming

Solving Part (B) first recommended

Challenge/encouragement:
If at least 30% of students submit Part (B) by 2/16, 11:59 PM,
we will bring cookies for the whole class!

Today:
- Stock gain problem (continued)
- Measuring efficiency
  - Practice
  - Theory
  - Models of computation
    - Random access machine (RAM)
    - Pointer machine
    - Python cost model

Reading:
CLRS 1, 2.1, 2.2, 3, 4.3, 4.4
Recall:

Stock gain problem: Given an array $A[0...n-1]$ of #s, find $0 \leq i^* \leq j^* < n$ s.t. $A[j^*] - A[i^*]$ is maximized.

Example:

$A = [15, 20, 3, 8, 16, 16, 12, 16, 16, 16]$

max gain = 13

Last time:

$\rightarrow$ Naïve $\Theta(n^2)$ time algorithm via brute force search

$\rightarrow$ Improved $\Theta(n \log n)$ time algorithm using divide & conquer

Note:

If $n = 10^6$

$\Theta(n^2) \rightarrow \sim 10^{12}$

$\Theta(n \log n) \rightarrow \sim 20 \cdot 10^6$

50K - improvement!

We got $\Theta(n \log n)$ algorithm.

Can we get an even faster one?

(This is the "eternal" question)
Indeed, we can do even better!

**Observation:** If we knew we went to sell at a particular day $j_0$ then we should buy at day $i_0$ s.t.

$$i_0 = \arg \min_{i \leq j_0} A[i]$$

In other words: Optimal gain $G^{*}[j_0]$ if selling on day $j_0$

$$G^{*}[j_0] := A[j_0] - \min_{i \leq j_0} A[i]$$

minimum of the prefix $A[0...j_0]$

$\Rightarrow$ (can compute in $\Theta(j_0+1)$ time)

How about computing best gains $G^{*}[j_0]$ for all possible $j_0$s and choosing the largest one?

**Problem:** Computing $G^{*}[j_0]$ for a single $j_0$ takes $\Theta(j_0+1)$ time

$\Rightarrow$ Total time to compute $G^{*}[j_0]$ for all $j_0$: Quadratic

$$\sum_{j_0=0}^{m-1} \Theta(j_0+1) \geq \sum_{j_0=\frac{m}{2}}^{m-1} \Theta(j_0+1) = \sum_{j_0=\frac{m}{2}}^{m-1} \Theta(m) \approx \frac{m}{2} \cdot \Theta(m) = \Theta(m^2)$$
What to do now?

**Key insight:** Don't need to compute each $G^*[j]$ from a scratch.

**Reuse computation!**

(V powerful idea - will talk more about it later)

**Note:** If $PMin[j] := \min_{i \leq j} A[i]$ then:

- $PMin[0] = A[0]$ (j = 0)
- $PMin[j] = \min(A[j], PMin[j-1])$ (j > 0)

Takes only $\Theta(1)$ time to compute.

**Resulting algorithm:**

$$\Theta(n) \begin{cases} 
  PMin[0] = A[0] & \text{# Compute all the} \\
  \text{For } 1 \leq j < n: & \text{# prefix minimas} \\
  PMin[j] = \min(A[j], PMin[j-1]) & \text{# Use prefix minimas} \\
  \text{max_gain = 0} & \text{# to compute best gain} \\
  \text{For } 1 \leq j < n: & \\
  \text{max_gain = max}(A[j] - PMin[j], \text{max_gain}) & \\
  \text{Return max_gain} & 
\end{cases}$$
We got an $\Theta(n)$ time algorithm!

Note: For $n = 10^6$, time $\approx 10^6$

1Mx improvement over $\Theta(n^2)$ algorithm
20x improvement over $\Theta(n \log n)$ alg.

Experiment with posted code!

Exercise: The algorithm above returns only the value of the best gain. How to modify it to get $(i^*, j^*)$ too?

(Challenge question: How to prove that $\Theta(n)$ time is optimal for this problem?)

Exercise: Design a "dual" variant of the above algorithm in which instead of looking at best gains from selling on a particular day $j_0$, we look at best gains from buying on a particular day $i_0$.

For a given day $i_0$, what is the best gain we can achieve if we want to buy on that day?
Measuring efficiency:

**Goal:** Design & analyze efficient algorithms

Efficient = Has "small" running time

Analyze = Determine (approximately) the running time

**Practice:** → Measure actual running time with wall clock

(can be sensitive to external factors (cache performance, other processes, etc.)

E.g., pypy - a better Python implementation can be 10x faster

→ Better: Use a profiler

(Does NOT depend on external factors)

Profiler: Provides usage statistics (# of min-sec/sec calls, etc.) for each function in the program

→ Can be used to keep track of how many times each block of code was executed
Theory:

Aims to understand:

- What about other values of \( n \)?
- What is the \textit{asymptotic} rate of growth?

\[
\begin{array}{c|c|c}
\text{Num} & \text{Sect} & \text{Formula} \\
1 & 43 & n^2 + [\sqrt{n}] \cdot 42 \\
2 & 46 & \Theta(n^2) \\
3 & 51 & \Theta(n^2) \\
4 & 100 & \Theta(n^2) \\
5 & 109 & \Theta(n^2) \\
\vdots & \vdots & \vdots \\
\end{array}
\]


Goal: Results of the form

\[ T(n) = \Theta(n^2) \]

or the like

\( \text{running time (e.g., in seconds)} \)  \( \text{input size} \)  \( \text{asymptotic notation} \)  \( \text{for } n = 0 \)

Weak,er goal:

\[ T(n) = O(n^2) \]

\( \text{asymptotic notation (for } \leq \text{)} \)

\( \text{(To get } \Theta(n^2) \text{ need to show } \Omega(n^2) \text{ too)} \)

\( \text{Asymptotic notation (for } \gg \text{)} \)
Input size: Needs to be defined for each problem, but there are common patterns

- Data
  - Array of length $n$  
  - $m \times n$ matrix
  - (large) integer $x \geq 0$
  - Graph $G = (E, V)$
  - Non-square $m \times n$ matrix

- Size
  - $m$
  - $m$ (even though there is $n^2$ #s)
  - $\log(x+2)$ (# of bits needed)
  - $|E|, |V|$ (edges, vertices)

Input size has two parameters

VERY Important:

- Many inputs can have the same size
- How to define $T(n)$ then?

In this class: We measure worst-case running time

$$T(n) = \max \text{ runtime over all inputs of size } n$$

- In principle, quite unrealistic but turns out to be a decent measure
We know how to measure efficiency
But: What determines it?

**Model of computation:** Specifies:
- What operations an algorithm is allowed
- Cost (time, space,...) of each operation
- Cost of algorithm = sum of op. costs

(1) Random Access Machine (RAM)

- Random Access Memory (RAM) modeled by a big ("infinite") array of words
- Θ(1) registers (each 1 word)
- In Θ(1) time, can
  - load word @ r_i into register r_j
  - compute (+,-,*,/,&,|) on registers
  - store register r_j into memory @ r_i

- **Word:** Assume basic objects fit in word (e.g. 64-bit int)
- Need $2 \log \text{(mem size)}$ for addressing
(2) **Pointer machine (PM):**

- PM can be simulated in RAM
- A useful abstraction (although weaker than RAM)
- RAM: "how stuff really works" (Assembler, C)
- PM: Object oriented programming

**PM Model:**

- Dynamically allocated **objects**
- Object has \(O(1)\) **fields**
- Field = word (e.g., int)
  or pointer to object / null
  (aka reference)

E.g., doubly-linked list:

```
+---+  +---+  +---+
| Val |  | Val |  | Val |
+-----+  +-----+  +-----+
| prev |  | prev |  | prev |
|      |  |      |  |      |
| null |  | null |  | null |
+-----+  +-----+  +-----+
| next |  | next |  | next |
+-----+  +-----+  +-----+
```

(3) **Python cost model:**

- A mix of RAM & PM abstraction
- "list" is actually an array \(\rightarrow\) RAM
  - \(L[i] = L[j] + 5\) \(\rightarrow\) \(O(1)\) time
- Object with \(O(1)\) attributes \(\rightarrow\) PM
  - \(x = x.next\) \(\rightarrow\) \(O(1)\) time
Warning! Not every "elementary" operation in Python has \( \Theta(1) \) cost.

To determine cost, imagine implementation in terms of RAM/PM.

**list:**

- \( L \).append(\( x \)) \( \implies \Theta(1) \) time
- To implement in RAM, use table doubling [Lecture 3]

\[
L = L1 + L2 \equiv \begin{cases} \Theta(1+|L1|+|L2|) \text{ time} \\ \end{cases}
\]

- \( L1 \).extend(\( L2 \)) \( \equiv \begin{cases} \text{ for } x \text{ in } L2:} \\
L1.\ append(x) \text{ \( \implies \} \Theta(|L2|) \end{cases}
\]

- \( \text{len}(L) \) \( \implies \Theta(1) \) time
- List stores its length in a field (and updates it)

- \( b = \text{in } L \equiv \begin{cases} \text{ for } y \text{ in } L:} \\
\text{ if } x == y \\
\text{ break } \\
\text{ else:} \\
\text{ b = False} \\
\text{ b = True \( \} \Theta(\text{index of } x \text{ in } L) \equiv \Theta(|L|) \text{ in the worst-case (e.g., if } x \notin L) \end{cases}
\]
\( \text{L.sort()} \rightarrow \Theta(1L1 \log 1L1) \text{ time} \)

\( \rightarrow \text{Why?} \rightarrow \text{[Lectures 3-4 & 7]} \)

\[ \text{dict: } D = \{ x_1:y_1, x_2:y_2, \ldots, x_n:y_n \} \]

\( \rightarrow \text{collection of (key, value) pairs} \)

\( \rightarrow D[\text{key}] = \text{val} \) \( \Theta(1) \) time (with high probability)

\( \rightarrow b = \text{key in } D \) \( \Theta(1) \) time too!

(Compare to \( \Theta(1L1) \) time for lists)

\( \rightarrow \text{key technique behind it: } \)

\( \text{Hashing } [\text{Lectures 8-10}] \)

\( \text{heapq: } \rightarrow \text{heappush & heappop } \rightarrow \Theta(\log n) \text{ time} \)

Via heaps \( [\text{Lecture 4}] \)

\( \text{Also: } \) See "Python cost model" for more

(Links on Stellar)