CS 457, Fall 2019

Drexel University, Department of Computer Science

Lecture 4

First Homework

- Due: Monday 10/7 (by midnight)
  - Late days

- No use of solutions available online!
- Collaboration allowed (only for this problem set)
- Typeset your solutions (LaTeX recommended but not required)
- My office hours are Tuesday 10am to noon
- Class recitation: Friday 1-3pm in room 1103
- Email me (with Vasilis cc:ed) with any homework-related questions
- Gradescope accounts

Maximum Subarray Problem

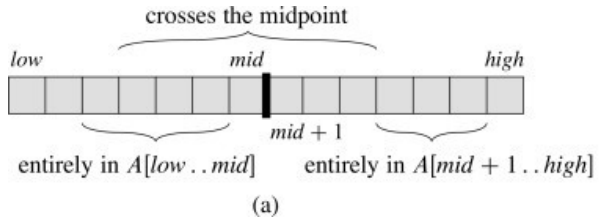
- You are given an array A of n numbers (both positive and negative)
  - Find a contiguous subarray with the maximum sum of numbers
  - In other words: find i, j such that $1 \leq i \leq j \leq n$ and maximize $\sum_{x=i}^{j} A[x]$
  - For example, consider the following array:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

- Divide and conquer
  - Divide: *Split the problem into smaller sub-problems of the same structure*
  - Conquer: *If sub-problem size is small enough, solve directly, o/w, solve sub-problems recursively*
  - Combine: *Merge the solutions of sub-problems into a solution of the original problem*

Maximum Subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i .. mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 .. j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```



(a)

Maximum Subarray Problem

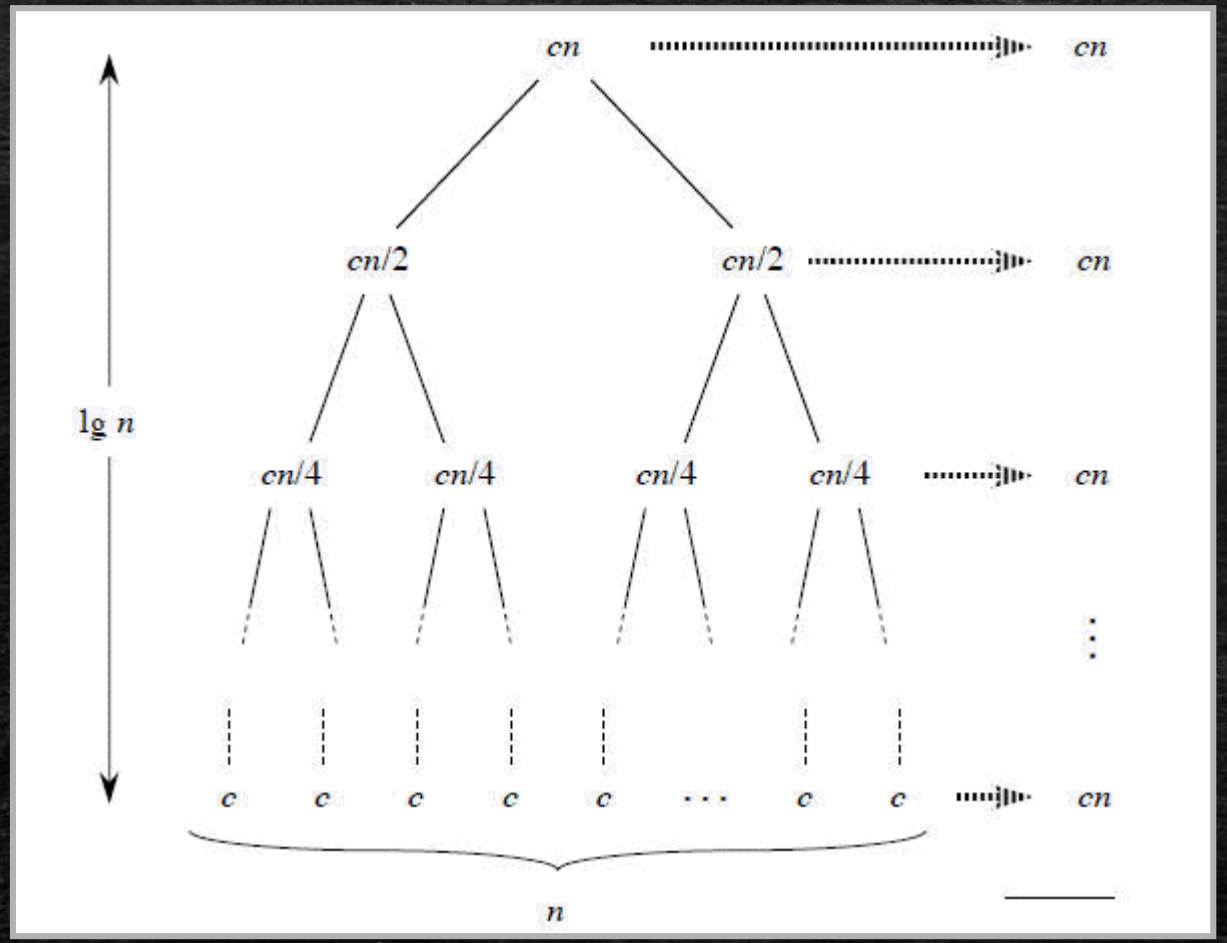
Divide-and-conquer procedure for the maximum-subarray problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Initial call: FIND-MAXIMUM-SUBARRAY$(A, 1, n)$

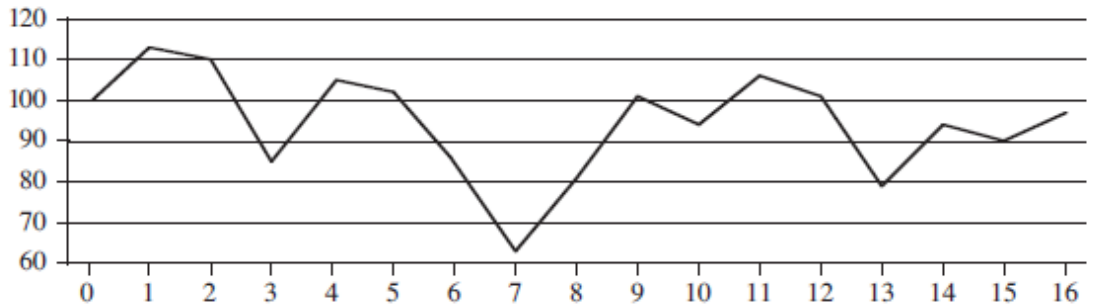
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

Recursion Tree



Profit Maximizing Stock Trade

- Input: n price points $(t_1, t_2, \ldots, t_n)$
- Output: (t_b, t_s) s.t. $0 \leq t_b < ts \leq n$, and $p(t_s) - p(t_b)$ is maximized



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- How would you solve this problem?
  - This problem can be easily reduced to the maximum subarray problem

Today's Lecture

- Analysis of recurrence equations
- More divide and conquer algorithms

Merge Sort

Sorting: Given a list A of n integers, create a sorted list of these integers

- Divide
  - *Split the problem into smaller sub-problems of the same structure*
  - Split the list A into two smaller lists of size n_1 and n_2
- Conquer
  - *If sub-problem size is small enough, solve directly, o/w, solve sub-problems recursively*
  - Sort the two smaller lists recursively using merge sort, unless their size is small
- Combine
  - *Merge the solutions of sub-problems into a solution of the original problem*
  - Merge the two sorted lists into one, and return the result

Merge Sort

MERGE-SORT (A, p, r)

```
1.      if p < r                                  // Check for base case
2.              q = ⌊(p + r)/2⌋                   // Divide step
3.              MERGE-SORT (A, p, q)              // Conquer step.
4.              MERGE-SORT (A, q + 1, r)          // Conquer step.
5.              MERGE (A, p, q, r)                // Conquer step.
```

To sort A[1 .. n], make initial call to MERGE-SORT (A, 1, n)

Merging Two Sorted Lists (running time)

MERGE (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
4. **for** i = 1 **to** n_1
5. L[i] = A[p + i – 1]
6. **for** j = 1 **to** n_2
7. R[j] = A[q + j]
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. i = 1
11. j = 1
12. **for** k = p **to** r
13. **if** L[i] ≤ R[j]
14. A[k] = L[i]
15. i = i + 1
16. **else**
17. A[k] = R[j]
18. j = j + 1

This needs time $\Theta(n_1 + n_2) = \Theta(r - p + 1)$

How about MERGE-SORT $(A, 1, n)$?

Running Time

- Recurrence equation for divide and conquer algorithms:
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$

- Recurrence equation for Merge Sort
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$

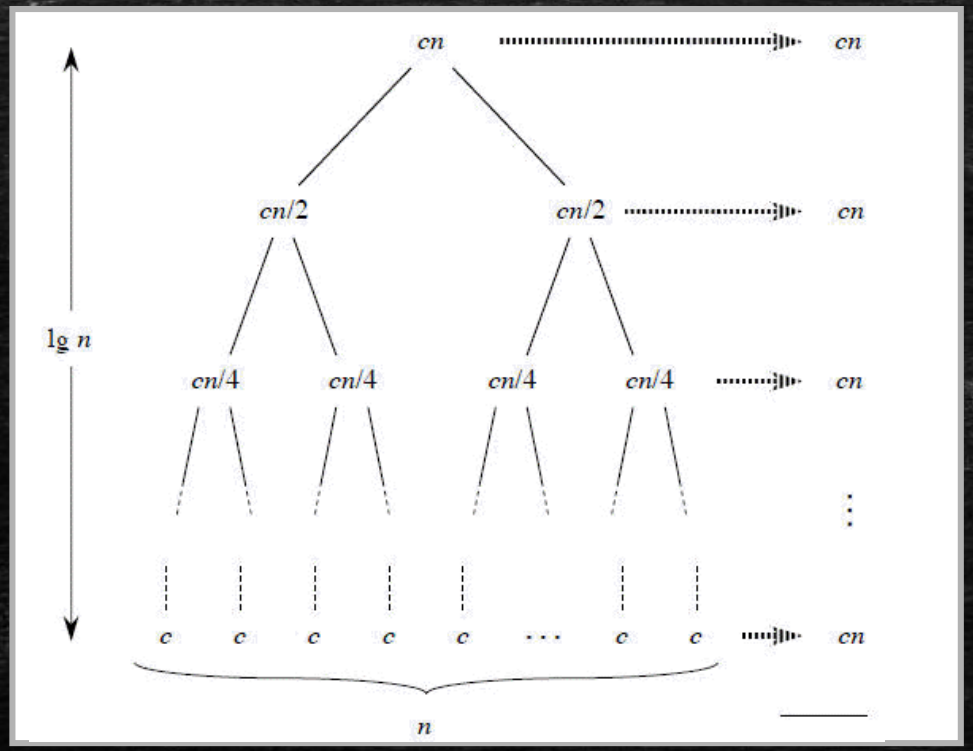
Methods for Solving Recurrences

Three methods:

1. Recursion-tree method
   - Covert into a tree and measure cost incurred at the various levels
2. Substitution method
   - Guess a bound and use mathematical induction to prove its correctness
3. Master method
   - Directly provides bounds for recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

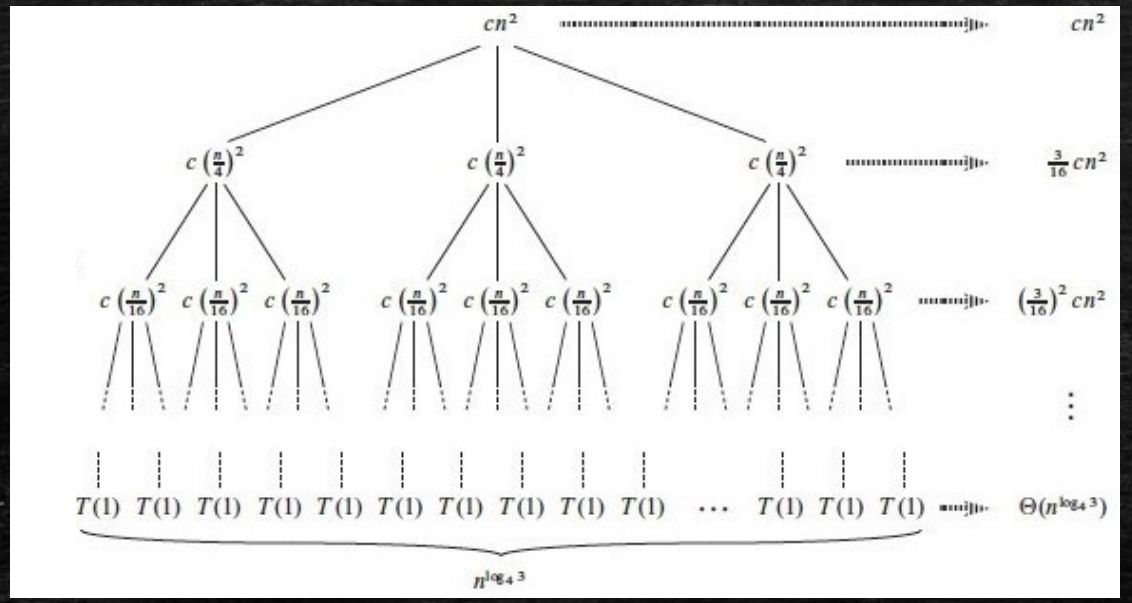
Recursion-Tree Method

- Recurrence equation for Merge Sort
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$



Recursion-Tree Method

- Recurrence equation
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2) & \text{otherwise} \end{cases}$



Substitution Method

1. Guess the form of the solution
2. Use mathematical induction to show that it works (for appropriate constants)

E.g., $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n) = 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$

Why wouldn't this work for $\boldsymbol{T(n) \leq cn}$ as well? (verify it!)

Guess that $T(n) = O(n \log n)$

Then, assume $T(n') \leq cn' \log n'$ for all $n' < n$, and show $T(n) \leq cn \log n$

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2[c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)] + n \\ &\leq cn \log(n/2) + n \\ &\leq cn \log n - cn \log 2 + n \\ &\leq cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant ε, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant ε, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n ,then $T(n) = \Theta(f(n))$

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n

1. $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$

$T(n) = 4T(n/3) + n \log n$

We apply the master theorem. If we let $a = 4$ and $b = 3$, we get $\log_b a = \log_3 4 > 1.25$. If we let $\epsilon = 0.01$, then $1 - \epsilon > 1.24$ and $f(n) = n \log n \in O(n^{1.24})$, so we can apply the first rule which yields $T(n) = \Theta(n^{\log_3 4})$.

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n
  1. $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$
  2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$

We first seek to apply the master theorem. In this case, $f(n) = n^2\sqrt{n} = n^{2.5}, a = 4, b = 2$ so we see that $n^{\log_b a} = n^{\log_2 4} = n^2$. Therefore, $f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon \in (0, 0.5]$, and our goal is to use the third rule of the master theorem. In order to do so though, we also need to verify that there exists a constant c such that $af(n/b) \leq cf(n)$, i.e., $4(n/2)^2\sqrt{n/2} = n^2\sqrt{n/2} \leq cn^2\sqrt{n}$. Since, e.g., $c = \sqrt{1/2}$ works, the third rule implies that:

$$T(n) = \Theta(n^2\sqrt{n}).$$

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n

1. $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$
2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
3. $T(n) = 3T\left(\frac{n}{3}\right) + n/\log n$

We first seek to apply the master theorem. If we let $f(n) = n/\log n, a = 3$, and $b = 3$, we see that $n^{\log_b a} = n^{\log_3 3} = n$. Since $n/\log n \in o(n)$, it is clear that the second and third rule of the master theorem cannot apply. But, as it happens, the first rule does not apply either, since $n/\log n \in \omega(n^{1-\epsilon})$ for any constant $\epsilon > 0$.

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n

1. $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$
2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
3. $T(n) = 3T\left(\frac{n}{3}\right) + n/\log n$

Using a recursion tree, we observe that its depth for this equation is $\Theta(\log n)$ and the total cost at depth d is $\frac{n}{\log\left(\frac{n}{3^d}\right)}$. This leads to:

$$T(n) \approx \sum_{d=1}^{\Theta(\log n)} \frac{n}{\log\left(\frac{n}{3^d}\right)} \approx n \sum_{d=1}^{\Theta(\log n)} \frac{1}{\log n - d} \approx n \sum_{d=1}^{\Theta(\log n)} \frac{1}{d} \approx \Theta(n \log\log n).$$

We therefore guess that $T(n) \in \Theta(n \log\log n)$, and prove this using the substitution method.