Exercise III, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

These problems are taken from various sources at EPFL and on the Internet, too numerous to cite individually.

1 Asymptotics and recursions

1 *(Previous exam question)* Give tight asymptotic bounds for the following recurrences (assuming that $T(1) = \Theta(1)$):

(i) $T(n) = 3T(n/3) + \Theta(1)$

(ii) $T(n) = 3T(n/3) + \Theta(n)$

(iii) $T(n) = 3T(n/3) + \Theta(n^2)$

(iv) $T(n) = T(n/5) + 2T(2n/5) + \Theta(n)$

(v) $T(n) = 25T(n/5) + \Theta(n^2)$

Solution:

(i) $\Theta(n)$ (Master Theorem)

(ii) $\Theta(n \log n)$ (Master Theorem)

(iii) $\Theta(n^2)$ (Master Theorem)

(iv) $\Theta(n \log n)$ (Substitution method)

(v) $\Theta(n^2 \log n)$ (Master Theorem)

2 *(previous exam question)*

Suppose you are choosing between the following five Divide-and-Conquer algorithms:

Algorithm A solves problems of size n by dividing (in constant time) them into two subproblems each of size $n/2$, recursively solving each subproblem, and then combining the solutions in $\Theta(n^3)$ time.

Algorithm B solves problems of size n by dividing (in constant time) them into nine subproblems each of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\Theta(n^2)$ time.

Algorithm C solves problems of size n by dividing (in constant time) them into ten subproblems each of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\Theta(n)$ time.

Algorithm D solves problems of size n by dividing (in constant time) them into eight subproblems each of size $n/2$, recursively solving each subproblem, and then combining the solutions in constant time.

Algorithm E solves problems of size n by dividing (in constant time) them into two subproblems each of size $n-1$, recursively solving each subproblem, and then combining the solutions in constant time.

What are the running times of each of these algorithms (in Θ notation), and which would you choose?

Solution:

Algorithm A has running time proportional to the recursion $T(n) = 2T(n/2) + \Theta(n^3)$ which, by the master theorem, is $\Theta(n^3)$.

Algorithm B has running time proportional to the recursion $T(n) = 9T(n/3) + \Theta(n^2)$ which, by the master theorem, is $\Theta(n^2 \lg n)$.

Algorithm C has running time proportional to the recursion $T(n) = 10T(n/3) + \Theta(n)$ which, by the master theorem, is $\Theta(n^{\log_3 10})$.

Algorithm D has running time proportional to the recursion $T(n) = 8T(n/2) + \Theta(1)$ which, by the master theorem, is $\Theta(n^3)$.

Algorithm E has running time proportional to the recursion $T(n) = 2T(n-1) + \Theta(1)$ which is $\Theta(2^n)$.

The running time of the algorithms are thus so that $B < C < A = D < E$. Therefore, we choose algorithm B.

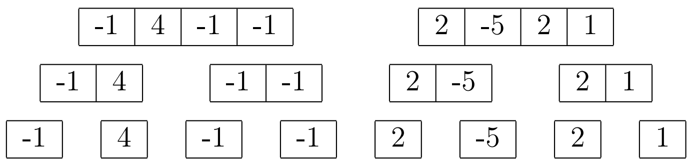
Maximum-Subarray Problem

3 In class we saw a divide-and-conquer algorithm for MAXIMUM SUBARRAY that runs in time $O(n \log n)$. Illustrate how it works by showing/explaining the divide, conquer and merge step of each subproblem when the algorithm is given the following input

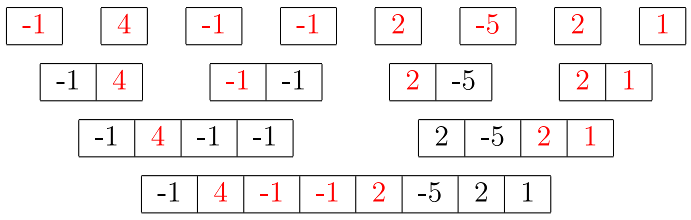
-1	4	-1	-1	2	-5	2	1

Solution:

1. Divide/Conquer Step: Divide the input into two smaller subproblems and then conquer these subproblems by recursively applying the procedure until we have singletons:



2. Combining or Merging Step: Let A_i be an array resulting at recursion level i. If i is the last recursion level (i.e A_i is a singelton), then the maximum subarray of A_i is just the whole array. Otherwise, the solution of the problem on A_i is the maximum among the solution of the left subproblem, the solution of right subproblem, and the maximum subarray that crosses the mid-point (this can be done in linear time). If we let red denote the current solution, we obtain:



Matrix Multiplication

4 (Exercise 4.2-1 in the book)
Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

Show your work.

Solution: Let $A = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}, B = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$

Denote by C the 2×2 output matrix.

1. Divide the matrices A, B and C into 1×1 matrices.
   $A_{11} = 1$, $A_{12} = 3$, $A_{21} = 7$, $A_{22} = 5$
   $B_{11} = 6$, $B_{12} = 8$, $B_{21} = 4$, $B_{22} = 2$

Page 3 (of ??)

2. Create the 10 matrices $S_1, S_2, ..., S_{10}$.
$S_1 = B_{12} - B_{22} = 8 - 2 = 6$,
$S_2 = A_{11} + A_{12} = 1 + 3 = 4$,
$S_3 = A_{21} + A_{22} = 7 + 5 = 12$,
$S_4 = B_{21} - B_{11} = 4 - 6 = -2$,
$S_5 = A_{11} + A_{22} = 1 + 5 = 6$,
$S_6 = B_{11} + B_{22} = 6 + 2 = 8$,
$S_7 = A_{12} - A_{22} = 3 - 5 = -2$,
$S_8 = B_{21} + B_{22} = 4 + 2 = 6$,
$S_9 = A_{11} - A_{21} = 1 - 7 = -6$,
$S_{10} = B_{11} + B_{12} = 6 + 8 = 14$.

3. Compute the 7 products $P_1, P_2, ..., P_7$.
$P_1 = A_{11} * S_1 = 1 * 6 = 6$,
$P_2 = S_2 * B_{22} = 4 * 2 = 8$,
$P_3 = S_3 * B_{11} = 12 * 6 = 72$,
$P_4 = A_{22} * S_4 = 5 * (-2) = -10$,
$P_5 = S_5 * S_6 = 6 * 8 = 48$,
$P_6 = S_7 * S_8 = (-2) * 6 = -12$,
$P_7 = S_9 * S_{10} = (-6) * 14 = -84$.

4. Add and subtract the P_i matrices to obtain the output C.
$C_{11} = P_5 + P_4 - P_2 + P_6 = 48 + (-10) - 8 + (-12) = 18$,
$C_{12} = P_1 + P_2 = 6 + 8 = 14$,
$C_{21} = P_3 + P_4 = 72 + (-10) = 62$,
$C_{22} = P_5 + P_1 - P_3 - P_7 = 48 + 6 - 72 - (-84) = 66$

Therefore $C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$

5 (Exercise 4.2-3 in the book)
How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

Solution: In class, we showed that we can compute the product of two $n \times n$ matrics in time $\Theta(n^{\lg 7})$ using Strassen's algorithm. Let's suppose we want to compute the product of two $n \times n$ matrices A and B where n isn't a power of 2, that is $2^i < n < 2^{i+1}$ for some $i > 0$.
In order to compute AB, we extend the dimension of A, B to $n' = 2^{i+1}$ by appending zeros to the colums and rows of A, B obtaining A', B'. Given that dimension of A' & B' is a power of 2, we can compute $A'B'$ by using Strassen's algorithm. Furthermore by computing the product $A'B'$, we can obtain AB.

$$A' \times B' = \begin{pmatrix} & & 0 & \cdots & 0 \\ & \mathbf{A} & 0 & \cdots & 0 \\ & & 0 & \cdots & 0 \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \end{pmatrix} \times \begin{pmatrix} & & 0 & \cdots & 0 \\ & \mathbf{B} & 0 & \cdots & 0 \\ & & 0 & \cdots & 0 \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \end{pmatrix} = \begin{pmatrix} & & 0 & \cdots & 0 \\ & \mathbf{AB} & 0 & \cdots & 0 \\ & & 0 & \cdots & 0 \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 \cdots 0 & 0 & \cdots & 0 \end{pmatrix}$$

The running time of Strassen's on $A'B'$ is $\Theta(n'^{\lg 7})$. Now, $n' = 2^{i+1} = 2 \times 2^i \leq 2n$. Hence, the running time will be $\Theta((2n)^{\lg 7})$ which is the same as $\Theta(n^{\lg 7})$.

6 (half *, Exercise 4.2-4 in the book)
What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

Note: $\lg x$ corresponds to the base 2 logarithm of x, i.e. $\lg x = \log_2 x$.

Solution: If you can multiply 3×3 matrices using k multiplications, then you can multiply $n \times n$ matrices by recursively multiplying $n/3 \times n/3$ matrices, in time $T(n) = kT(n/3) + \Theta(n^2)$. Using the master method to solve this recurrence, consider the ratio of $n^{\log_3 k}$and n^2:

- If $\log_3 k = 2$, case 2 applies and $T(n) = \Theta(n^2 \lg n)$. In this case, $k = 9$ and $T(n) = o(n^{\lg 7})$.
- If $\log_3 k < 2$, case 3 applies and $T(n) = \Theta(n^2)$. In this case, $k < 9$ and $T(n) = o(n^{\lg 7})$.
- If $\log_3 k > 2$, case 1 applies and $T(n) = \Theta(n^{\log_3 k})$. In this case, $k > 9$. $T(n) = o(n^{\lg 7})$ when $\log_3 k < \lg 7$, i.e., when $k < 3^{\lg 7} \approx 21.85$. The largest such integer k is 21.

Thus, $k = 21$ and the running time is $\Theta(n^{\log_3 k}) = \Theta(n^{\log_3 21}) = O(n^{2.80})$ (since $\log_3 21 \approx 2.77$).

More general questions

7 *(previous exam question)* Consider the procedure Power that takes as input a number a, a non-negative integer n and returns a^n:

```
Power($a, n$)
1. if $n = 0$
2.     return 1
3. if $n = 1$
4.     return a
5. $q = \lfloor \frac{n}{4} \rfloor + 1$
6. return Power($a, q$)$\cdot$ Power($a, n - q$)
```

7a *(10 pts)* Let $T(n)$ be the time it takes to invoke Power(a, n). Then the recurrence relation of $T(n)$ is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \text{ or } n = 1, \\ T(\lfloor n/4 \rfloor + 1) + T(n - \lfloor n/4 \rfloor - 1) + \Theta(1) & \text{if } n \geq 2. \end{cases}$$

Prove that $T(n) = O(n)$ **using the substitution method**.

In your proof you may ignore the floor function, i.e., you can replace $\lfloor n/4 \rfloor$ by $n/4$.

Solution:

We shall show that $T(n) = O(n)$.

Claim 1.1 *There exist positive constants b, b' and n_0 such that $T(n) \leq bn - b'$ for all $n \geq n_0$.*

Proof. As always in the substitution method we do not need to worry about the base case as we can always select b large enough so that the base case holds.

Now consider the inductive step, i.e., assume statement true for all $k < n$ and we prove it for n:

$$\begin{aligned} T(n) &\leq T(n/4 + 1) + T(3n/4 - 1) + c \qquad \text{for some absolute constant } c \\ &\leq b(n/4 + 1) - b' + b(3n/4 - 1) - b' + c \\ &= bn - 2b' + c \qquad \text{selecting } b' \text{ to be a greater constant than } c \text{ gives us} \\ &\leq bn - b', \end{aligned}$$

as required.

□

7b *(15 pts)* **Design** and **analyze** a modified procedure FASTPOWER(a, n) that returns the same value a^n but runs in time $\Theta(\log n)$.

A solution that only works when n is a power of 2, i.e., $n = 2^k$ for some integer $k \geq 0$, gives partial credits.

(Note that a^n is *not* a basic instruction and should therefore not be used.)

Solution:
Note that

$$a^n = \begin{cases} 1 & \text{if } n = 0, \\ (a^{\lfloor n/2 \rfloor})^2 & \text{if } n > 0 \text{ is even}, \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{if } n > 0 \text{ is odd}. \end{cases}$$

This motivates the following Divide-and-Conquer solution which modifies the previous solution:

```
FASTPOWER($a, n$)
1. if $n = 0$
2.         return 1
3. $tmp$ =FASTPOWER($a$, $\lfloor n/2 \rfloor$)
4. if $n$ is even
5.         return $tmp \cdot tmp$
6. else ($n$ is odd)
7.         return $tmp \cdot tmp \cdot a$.
```

To analyze this, let $T(n)$ be the time it takes to execute FASTPOWER(a, n). Note that the divide and combine part both takes time $\Theta(1)$. Further we conquer one subproblem of size $\lfloor n/2 \rfloor \approx n/2$. Therefore we have that $T(n) = T(n/2) + \Theta(1)$ and $T(0) = \Theta(1)$. By the Master method we thus have that $T(n) = \Theta(\log n)$ as required.

8 ($*$, Exercise 4.1-5 in the book) Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \,.\,.\, j]$, extend the answer to find a maximum subarray ending at index $j + 1$ by using the following observation: a maximum subarray of $A[1 \,.\,.\, j + 1]$ is either a maximum subarray of $A[1 \,.\,.\, j]$ or a subarray of the form $A[i \,.\,.\, j + 1]$, for some $1 \leq i \leq j + 1$. Determine a maximum subarray of the form $A[i \,.\,.\, j + 1]$ in constant time based on knowing a maximum subarray ending at index j.

Solution: Let $maxSub(A[i \cdots j])$ be the maximum subarray of $A[i \cdots j]$ and $sum(A[i \ldots j])$ be the summation of the entries in array A from index i to index j. Then $maxSub(A[1 \cdots n])$ can be computed as follows.

$$maxSub(A[1 \cdots n]) = \max(maxSub(A[1 \cdots n-1]), \max_{i=1}^{n} sum(A[i \cdots n]))$$

That is, maximum subarray of $A[1 \cdots n]$ either includes $A[n]$ (2^{nd} part in the above recurrence) or not ($maxSub(A[1 \cdots n]) = maxSub(A[1 \cdots n-1])$).

```
input: array A of length n
max_so_far=-infinity
max_ending_here=-1
for i in 1:n
if(max_ending_here<0)
   max_ending_here=A[i]
else
   max_ending_here=max_ending_here+A[i]
endif
 max_so_far=max(max_ending_here, max_so_far)
endfor
output: max_so_far
```

Note the $-\infty$ above can be replaced with the smallest number in the array (this takes linear time). Alternatively,

```
input: array A of length n
max_so_far=0
max_ending_here=0
for i in 1:n
   max_ending_here=max(max_ending_here+a[i],0)
   max_so_far=max(max_ending_here, max_so_far)
output: max_so_far
```

Note that the above algorithm (second one) assumes that the maximum subarray has a non-negative sum. In order to take care of the case when the maximum subarray has a negative sum (this exists when the array is only made of negative numbers), we can check if this is the case and just output the maximum negative element in linear time.
Both algorithms run in linear time since each line takes $\Theta(1)$ and each line us run at most n times because of the loop, thus $\Theta(n)$ total running time. Hence, we obtain a linear time, non-recursive algorithm solving the maximum subarray problem.