| **CS 3510: Design and Analysis of Algorithms** | Georgia Tech |
|---|---|
| Exam 1 | |
| *Instructor: Shahrokh Shahi* | *Summer 2022* |

| **GT Username:** | **Full Name:** |
|---|---|
| | |

# Instructions:

1. Write your **name** and **GT username** on each page very clearly. Then, complete the exam.
2. This exam is closed-book, and collaboration is NOT permitted.
3. You are allowed to use **one sheet of notes**, i.e., both sides of a letter-sized paper, during the exam.
4. No calculator is required.
5. You have **80 minutes** to complete this exam.
6. It is recommended to read all the questions before starting. Please read the questions carefully. Misunderstanding the question is not a valid excuse for losing points.
7. If you find it necessary, make reasonable assumptions but make sure to state them clearly.
8. You can use the back of each sheet as scratch paper.
9. Write your solution is the space provided. In case you need more space, you can use back of the same sheet, and make a notation on the front of the sheet.
10. The exam has 50 points in total.

**Good luck!**

| Number | Problem | Points | Grade |
|---|---|---|---|
| 1 | Asymptotic Notations | 12 | |
| 2 | Master Theorem | 10 | |
| 3 | Divide-and-Conquer: Tri-Merge-Sort | 8 | |
| 4 | Divide-and-Conquer: Binary Search | 10 | |
| 5 | Dynamic Programming | 10 | |

| GT Username: | Full Name: |
| --- | --- |
| | |

# 1 Asymptotic Notations [12 pts]

(a) (5 pts) For each pair of functions $f$ and $g$, choose one of $f \in O(g), f \in \Theta(g), f \in \Omega(g)$ that best describes their relative asymptotic growth. No justification is required.

- $f = \log(n^3)$, $g = 100\log(n)$
  Sol: $\Theta$ (1 pt)

- $f = n^4$, $g = (n \log n)^3 + n^2$
  Sol: $\Omega$ (1 pt)

- $f = n^{1000}$, $g = 1.5^n$
  Sol: $O$ (1 pt)

- $f = 2^n$, $g = (\frac{5}{2})^n$
  Sol: $O$ (1 pt)

- $f = (n+3)^3$, $g = 100n^3 - n$
  Sol: $\Theta$ (1 pt)

(b) (3 pts) Give the mathematical definition of $f(n) \in \Omega(g(n))$ and provide an example.

$f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n_0$.

or

$f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$, $f(n) \geq cg(n) \geq 0$

Ex. $f(n) = n^2, g(n) = n \log n \Rightarrow f(n) = \Omega(g(n))$

(c) (4 pts) Assume you have functions $f$ and $g$, such that $f(n) \in O(g(n))$. For the following statement, tell whether it is true or false, and give a proof (if it is true) or a counterexample (if it is false).

$$\boxed{\texttt{if } g(n) \in O(h(n))\texttt{, then } f(n) \in O(h(n)).}$$

**True (Transitivity of asymptotic growth rate)**

We are given $f(n) = O(g(n))$; therefore, for some constants $c_1 > 0$ and $n_1 \geq 0$, we have $f(n) \leq c_1 g(n)$ for all $n \geq n_1$. Moreover, due to $g(n) = O(h(n))$, for some $c_2 > 0$ and $n_2 \geq 0$, we have $g(n) \leq c_2 h(n)$ for all $n \geq n_2$. Now, consider any number $n$ that is at least as large as both $n_1$ and $n_2$. We have $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$, and so $f(n) \leq c_1 c_2 h(n) = ch(n)$, where $c = c_1 c_2 > 0$, for all $n \geq \max(n_1, n_2)$. Thus, $f(n) = O(h(n))$.

| GT Username: | Full Name: |
|---|---|
| | |

# 2 Master Theorem [10 pts]

## 2.1 Solve the following recurrence relations using the Master Theorem and give the tightest bound in terms of $\Theta$. Also, state whether the computational cost is dominated at the leaves, the root or equally distributed at all levels of the corresponding recursion tree.

(a) (3 pts) $T(n) = 9T(n/3) + \Theta(n^2)$
$a = 9, b = 3, d = 2 \Rightarrow a = b^d \Rightarrow T(n) = \Theta(n^2 \log n)$
equally distributed

(b) (3 pts) $T(n) = 5T(n/4) + \Theta(\frac{1}{2}n + 5)$
$a = 5, b = 4, d = 1 \Rightarrow a > b^d \Rightarrow T(n) = \theta(n^{\log_4 5})$
dominated at the leaves

## 2.2 (4 pts) For the following divide-and-conquer program, give the recurrence relation describing the running time and apply the Master Theorem to calculate the running time.

```
def func(n):
    if n==0: stop

    func(n/3)

    func(n/3)

    do_something in O(n)
```

Recurrence relation: $T(n) = 2T(n/3) + O(n)$
Using master theorem: $a = 2, b = 3, d = 1$, and $a < b^d$. Therefore, $T(n) = \Theta(n^d) = \Theta(n)$

| GT Username: | Full Name: |
| --- | --- |
| | |

## 3 Divide-and-Conquer: Tri-Merge-Sort [8 pts]

We would like to build a more advanced version of the `Merge-Sort` algorithm in which at each step the array will be divided into three sub-arrays. Answer the following questions:

(a) (4 pts) The merging step in the `Merge-Sort` algorithm (discussed in lectures) combines two sorted sub-arrays in linear time. Now, suppose we have three sorted sub-arrays A, B, and C, each of length $n/3$, and we want to merge them into a single sorted array $S$ of length $n$ containing all elements of these three sub-arrays. Design an algorithm that can combine these three sorted sub-arrays in linear time. Describe your algorithm in words or pseudocode.

This is similar to the merging step in the normal merge-sort algorithm – this extension was also discussed in lecture 2 (see demo code in lecture 2).

```
def tri_merge(A, B, C):
    n = len(A)
    merged = [0]*(3*n) # place-holder for the output
    p1 = p2 = p3 = i = 0
    INF = float("inf")

    while i < 3*n:
        a = A[p1] if p1 < n else INF
        b = B[p2] if p2 < n else INF
        c = C[p3] if p3 < n else INF
        if a <= b and a <= c:
            merged[i] = a
            p1 += 1
        elif b <= a and b <= c:
            merged[i] = b
            p2 += 1
        else:
            merged[i] = c
            p3 += 1
        i += 1
    return merged
```

(b) (4 pts) Consider the `Tri-Merge-Sort` algorithm in which the given array is divided into three equal length sub-arrays. Each sub-array is sorted recursively, and then, the three sorted sub-arrays are combined using your linear time algorithm from part (a). Provide the recurrence relation describing the running time of the `Tri-Merge-Sort` and apply the Master Theorem to obtain the time complexity of this algorithm.

$\boxed{T(n) = 3T(n/3) + O(n)} \Rightarrow a = 3, b = 3, d = 1 \Rightarrow a = b^d$

$\Rightarrow \boxed{T(n) = \Theta(n \log n)}$

| GT Username: | Full Name: |
|---|---|
| | |

# 4 Divide-and-Conquer: Binary Search [10 pts]

Given a sorted array $A = [a_1, a_2, \ldots, a_n]$ including all the integers in the range $\{1, 2, \ldots, n-1\}$ exactly once, expect for one of them which appears twice. Design a divide and conquer algorithm to find the only repeated element.

Example 1: $\boxed{\text{repeated element} = 3}$

$$A = [1, 2, 3, 3, 4, 5]$$

Example 2: $\boxed{\text{repeated element} = 1}$

$$A = [1, 1, 2, 3, 4, 5, 6, 7]$$

(a) (5 pts) Explain your algorithm in words, and justify its correctness.

**Algorithm:** We proceed in a binary search fashion: for interval $[lo, hi]$, let $mid \leftarrow \lfloor \frac{lo+hi}{2} \rfloor$. Check $A[mid] - mid$:

- If $A[mid] - mid = 0$, it means that the repeated element is in the right side (has not appeared yet). So, set $lo = mid + 1$ and repeat.
- If $A[mid] - mid = -1$, it means either this element is the repeated one or the repeated one is on the left side (it has appeared earlier). So,
  - if $A[mid] = A[mid-1]$ or $lo = hi$, return $A[mid]$
  - otherwise, set $hi = mid$ and repeat

**Explanation:** The array is sorted and contains the numbers $1, 2, \ldots, n-1$. Therefore, $A[i] - i$ must be equal to either 0 or $-1$ for all indexes, where the former happens when the repeated element has not appeared yet, and the latter happens after the appearance of the repeated element. (The repeated element will be the first time we have $A[i] \neq i$).
The binary search step is justified by those observations: if $A[mid] = mid$ then all indexes $i < mid$ must satisfy $A[i] = i$, and we can safely iterate to the second half of the array, as the repeated element must be there. If $A[mid] = mid - 1$ there is one checking step needed before iterating as we must verify if the repeated element landed at indexes $mid - 1, mid$. If not, we iterate to the first half since the repeated element must occurs before index $mid$ in order for the shift in the value to occurs. Every iteration is guarantee to contain an index $t$ with the property $A[t] = t - 1$, so we are guarantee to end while performing the first step which return the repeated element.

(b) (3 pts) Provide the pseudocode describing your algorithm.

**Algorithm 1:** `FindRepeated`

**Input:** $A = \{a_1, a_2, ..., a_n\}$
**Result:** the repeated element $a_i$

```
lo ← 0
hi ← n

while (lo < hi) do
    mid ← ⌊(lo + hi)/2⌋
    if (A[mid]==mid) then
        lo ← mid + 1
    else if A[mid]==A[mid-1] then
        return A[mid]
    else
        hi ← mid
    end
end

return A[lo]
```

Note: Other implementations, e.g., recursive approach is also accepted.

```python
# Python implementation
def find_repeated(A):
    n = len(A)
    l, r = 0, n-1

    while l < r:
        m = (l+r)//2

        if A[m]-m == +1: # 0-indexed
            l = m+1
        elif A[m]==A[m+1]:
            return A[m]
        else:
            r = m

    return A[l]
```

(c) (2 pts) Analyze the running time of your algorithm using the Master Theorem.

$\boxed{T(n) = T(n/2) + O(1)} \Rightarrow a = 1, b = 2, d = 0 \Rightarrow a = b^d$

$\Rightarrow \boxed{T(n) = \Theta(\log n)}$

| GT Username: | Full Name: |
|---|---|
| | |

## 5 Dynamic Programming: Maximum Paired Sum [10 pts]

Consider an array of $n$ integer numbers $A = [a_1, a_2, \ldots, a_n]$. Design an algorithm to find the maximum `Adjacent-Pair-Product-Sum`, which is defined as the largest value that can be obtained by multiplying adjacent elements in the array and then add them together. Each element can be paired with at most one of its immediate neighbors, but it is also allowed to be left alone.

Example 1: $A = [1, 2, 3, 1]$
Maximum `Adjacent-Pair-Product-Sum` $= 1 + (2 \times 3) + 1 = 8$

Example 2: $A = [2, 2, 1, 3, 2, 1, 2, 2, 1, 2]$
Maximum `Adjacent-Pair-Product-Sum` $= (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2 = 19$.
But another `Adjacent-Pair-Product-Sum` that is not optimal is $2 + (2 \times 1) + (3 \times 2) + 1 + (2 \times 2) + 1 + 2 = 18$.

(a) (1 pt) Compute the largest `Adjacent-Pair-Product-Sum` of $A = [1, 4, 3, 2, 3, 4, 2]$

Maximum `Adjacent-Pair-Product-Sum` $= 1 + (4 \times 3) + 2 + (3 \times 4) + 2 = 29$

(b) (4 pts) Discuss the optimal substructure of the `Adjacent-Pair-Product-Sum` problem, and give the recurrence relation including the base case(s).
*You can define $OPT[i]$ as the largest Adjacent-Pair-Product-Sum of the first $i$ elements, $a_1, \ldots, a_i$.*

Define $OPT[i]$ as the largest `Adjacent-Pair-Product-Sum` of array $A_i = [a_1, a_2, \ldots, a_i]$. Then, the optimum solution of the current step can be constructed using the optimum solutions of the previous steps. Accordingly, at step $i$, we have two options for using $a_i$:

(1) We can treat it as an alone number, where in this case, the optimum solution at step $i$ is equal to $OPT[i-1] + a_i$, or

(2) We can pair the current element $a_i$ with the previous element $a_{i-1}$, where the optimum solution will obtain by adding their multiplication $a_i \times a_{i-1}$ to the optimum solution obtained at step $OPT[i-2]$.

The optimum solution at current step $i$ (which is the largest `Adjacent-Pair-Product-Sum` of array $A_i = [a_1, a_2, \ldots, a_i]$) is the maximum of these two values:

$$OPT[i] = \max\{OPT[i-1] + a_i, OPT[i-2] + a_i \times a_{i-1}\}$$

where the base cases are $OPT[0] = 0$, $OPT[1] = a_1$.

| GT Username: | Full Name: |
| --- | --- |
| | |

(c) (3 pts) Give the pseudocode of a bottom-up or top-down implementation of the dynamic programming algorithm using the recurrence relation from part (a).

```
# Python implementation
def paired_sum(A):
    n = len(A)
    opt = [0] * (n+1)
    opt[1] = A[0]

    for i in range(2, n+1):
        opt[i] = max(opt[i-1]+A[i-1], opt[i-2]+A[i-1]*A[i-2])

    return opt[n]
```

(d) (2 pts) Analyze the time and space complexity of your algorithm.

Time: $O(n)$, Space: $O(n)$

Since we only need the solution of the last two subproblems, the space complexity can be reduced to $O(1)$.