recursion	**April 16, 2025**

Homework 2

Spring 2025	*COSC 31: Algorithms*

1. **Master Theorem**
   Use the Master theorem to solve the following recurrrences. Be sure to state which case of the master method applies and why.

   (a) $T(n) = 16T(\frac{n}{4}) + n^2$

   Answer

   To verify master theorem applies, we compare $f(n) = n^2$ to $n^{\log_b a}$, where $a = 16 \geq 1$, and $b = 4 > 1$

$$
\begin{aligned}
f(n) &= n^2 \\
n^{\log_b a} &= n^{\log_4 16} = n^2 \\
f(n) &= \Theta(n^{\log_4 16} \lg^k n) \quad k = 0 \\
f(n) &= \Theta(n^{\log_4 16} \lg^0 n) \\
&= \Theta(n^{\log_4 16}) \\
&= \Theta(n^2)
\end{aligned}
$$

   This fits into case 2 where $k = 0$. Therefore, $T(n)$ is:

$$
\begin{aligned}
T(n) &= \Theta(n^{\log_4 16} \lg^{0+1} n) \\
&= \Theta(n^2 \lg n)
\end{aligned}
$$

   (b) $T(m) = 7T(\frac{m}{3}) + m^2$

   Answer

   $a = 7, b = 3$

$$
\begin{aligned}
f(m) &= m^2 \\
m^{\log_3 7} &= m^{\log_3 7 + \epsilon}
\end{aligned}
$$

   We see that $\log_3 7 < 2$, hence we have $\epsilon = 2 - \log_3 7$. Therefore, this fits into case 3, and $f(m)$ is:

$$
f(n) = \Omega(m^{(\log_3 7) + \epsilon}); \quad \epsilon > 0
$$

Checking the regularity condition we have:

$$
\begin{aligned}
7 \cdot f(\frac{m}{3}) &= 7 \cdot \left(\frac{m}{3}\right)^2 \\
&= \frac{7}{9} m^2 \\
&\leq c \cdot m^2
\end{aligned}
$$

The condition holds for $\frac{7}{9} < c < 1$, therefore we can find a c that satifises the condition. $T(m)$ is:

$$T(n) = \Theta(m^2)$$

(c) $T(n) = 2T(\frac{n}{4}) + \sqrt{n} \lg^2 n$

Answer

$a = 2, b = 4$

$$
\begin{aligned}
f(n) &= \sqrt{n} \lg^2 n \\
n^{\log_4 2} &= \sqrt{n}
\end{aligned}
$$

This fits into case 2 where $k = 2$. Therefore, we have:

$$
\begin{aligned}
f(n) &= \Theta(n^{\log_4 2} \lg^2 n) \\
T(n) &= \Theta(n^{\log_4 2} \lg^{2+1} n) \\
&= \Theta(n^{\frac{1}{2}} \lg^3 n) \\
&= \Theta(\sqrt{n} \lg^3 n)
\end{aligned}
$$

2. **Recursion Tree**
Solve the following recurrences by the recursion tree method. Show your steps. (Your answer should be as tight as possible. For instance, if the answer is $\mathcal{O}(n^2)$, showing a bound of $\mathcal{O}(n^3)$ wont́ get any points.)

(a) $T(n) = T(n-2) + \mathcal{O}(n^2)$

Answer

Below is the recursion tree pattern with k levels

$$
\begin{array}{ll}
k = 0 & T(n) \leq T(n-2) + dn \\
k = 1 & T(n-2) \leq T(n-4) - d(n - 2 \cdot 1)^2 \\
k = 2 & T(n-4) \leq T(n-4) - d(n - 2 \cdot 2)^2 \\
k = 3 & T(n-6) \leq T(n-6) - d(n - 2 \cdot 3)^2 \\
k = 4 & T(n-8) \leq T(n-8) - d(n - 2 \cdot 4)^2
\end{array}
$$

We see the pattern of the recursion tree for the k^{th} is:

$$T(n - 2 \cdot k) \leq T(n - 2 \cdot k) + d(n - 2 \cdot k)^2$$

Assuming, $T(0)$ is the base case, we have $n - 2 \cdot k = 0$, hence $k = \frac{n}{2}$ levels of tree. Therefore, the time complexity is:

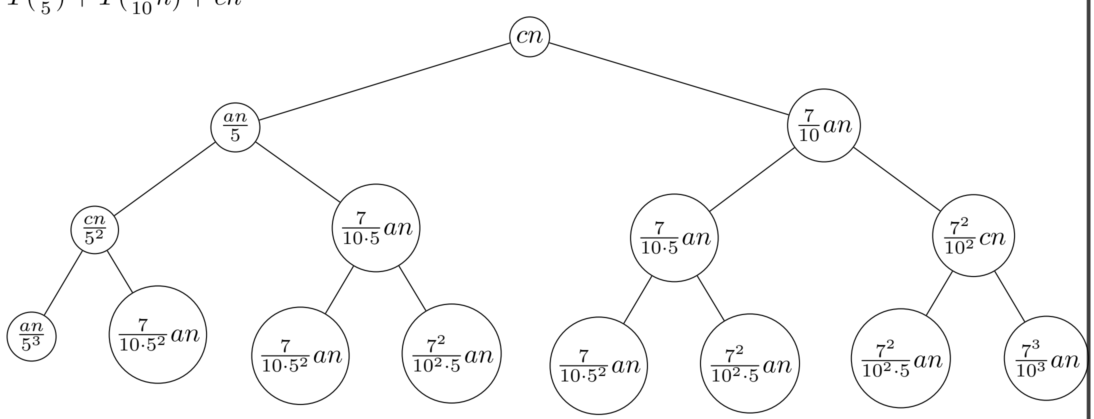
$$\begin{aligned}
T(n) &= \sum_{i=0}^{\frac{n}{2}} d(n-2\cdot i)^2 \\
&\leq \sum_{i=0}^{\frac{n}{2}} dn^2 \quad \text{because } (n-2i) < n \text{ hence } (n-2i)^2 \leq n^2 \text{ for all i} \\
&= dn^2 \cdot \sum_{i=0}^{\frac{n}{2}} 1 \\
&= (\frac{n}{2}+1)\cdot dn^2 \\
&\leq n \cdot dn^2 \\
&= dn^3 \\
&= O(n^3)
\end{aligned}$$

We see that $T(n) = O(n^3)$ for $n_0 = 1, c = d$.

(b) $T(n) = T(\frac{n}{5}) + T(\frac{7}{10}n) + O(n)$

Answer

Below is the tree representation of the recursion tree with k levels for $T(n) \leq T(\frac{n}{5}) + T(\frac{7}{10}n) + cn$



We see combining pattern below:

$k = 1 \quad an\left(\frac{1}{5} + \frac{7}{10}\right)^1$

$k = 2 \quad \frac{an}{5^2} + 2\cdot\frac{7}{10\cdot 5}cn + \frac{7^2}{10^2}cn \equiv cn\left(\frac{1}{5} + \frac{7}{10}\right)^2$

$k = 3 \quad \frac{an}{5^3} + 3\cdot\frac{7}{10\cdot 5^2}an + 3\cdot\frac{7^2}{10^2\cdot 5}an + \frac{7^3}{10^3}an \equiv an\left(\frac{1}{5} + \frac{7}{10}\right)^3$

$k = \quad an\left(\frac{9}{10}\right)^k$

Since the combining time at each level is a Big-O of cn, we choose $\frac{7}{10}$ reduction of the tree. The number of levels of the tree k is

$$n = \left(\frac{7}{10}\right)^k$$
$$\lg n = -k \lg \frac{7}{10}$$
$$k = \frac{1}{\lg \frac{10}{7}} \cdot \lg n$$

Let $d = \frac{1}{\lg \frac{10}{7}}$, we have $k = d \cdot \lg n$. The time complexity is:

$$T(n) = an \sum_{i=0}^{k} \left(\frac{9}{10}\right)^i \tag{1}$$
$$= an \cdot \frac{1 - \frac{9}{10}^{k+1}}{1 - \frac{9}{10}} \tag{2}$$
$$\leq an \cdot \frac{1}{\frac{1}{10}} \tag{3}$$
$$= 10an \tag{4}$$
$$= \mathcal{O}(n) \tag{5}$$

We see that $T(n) = \mathcal{O}(n)$ for $n_0 = 1, c = 10a$.

(c) $T(n) < \sqrt{n}T(\sqrt{n}) + n$

Answer

$k = 1 \quad T(n^{\frac{1}{2}}) < \sqrt{n}T(n^{\frac{1}{2^2}}) + n^{1-\frac{1}{2}} \cdot an^{\frac{1}{2}}$ we have $\sqrt{n}$ nodes, each having a combine time of $\sqrt{n}$

$k = 2 \quad T(n^{\frac{1}{2^2}}) < \sqrt{n}T(n^{\frac{1}{2^3}}) + n^{1-\frac{1}{2^2}} \cdot an^{\frac{1}{2^2}}$

$k = 3 \quad T(n^{\frac{1}{2^3}}) < \sqrt{n}T(n^{\frac{1}{2^4}}) + n^{1-\frac{1}{2^3}} \cdot an^{\frac{1}{2^3}}$

$k^{th} \quad T(n^{\frac{1}{2^k}}) < \sqrt{n}T(n^{\frac{1}{2^k}}) + an$

We see that each level adds up to n. The base case would be $n = 2$, and the number of levels k is:

$$n^{1/2^k} = p; \quad p \text{ is a constant}$$
$$\frac{1}{2^k} \cdot \log_2 n = \log_2 p$$
$$\frac{\lg n}{\lg p} = 2^k$$
$$k = \frac{\lg \lg n}{\lg p}$$

Let $d = \frac{1}{\lg p}$, then $k = d \cdot \lg \lg n$

Therefore, the number of levels is $k = d \cdot \lg \lg n$. The time complexity is:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{k} an \\
&= an \sum_{i=0}^{k} 1 \\
&= an \cdot \sum_{i=0}^{k} 1 \\
&= an \cdot (k+1) \cdot 1 \\
&= an \cdot (d \cdot \lg \lg n + 1) \\
&\leq ad \cdot n \lg \lg n + n \lg \lg n \\
&= n \lg \lg n (ad + 1) \\
&= \mathcal{O}(n \lg \lg n)
\end{aligned}$$

We see that $T(n) = \mathcal{O}(n \lg \lg n)$ for $n_0 = 1, c = ad + 1$. If assume that $2^k = 2^1$ then, $d = 1$, but I am accounting for case where n doesn't decay for 2^1.

3. **Local Minimum**
   Given an $n > 1$ and an array $A[1 \ldots n]$ of distinct integers, an index $2 \leq j \leq n-1$ is a *local minimum* if $A[j] < A[j-1]$ and $A[j] < A[j+1]$. 1 is a local minimum if $A[1] < A[2]$, and n is a local minimum if $A[n] < A[n-1]$.

   (a) Design a recursive $O(\log n)$-time algorithm to find some local minimum.

Answer

Algorithm FindLocalMinimum

```
Function FindLocalMinimum(A[1...n]):
    if A[1] < A[2] then
        return 1 ;                                         // 1 is a local minimum
    end
    if A[n] < A[n-1] then
        return n ;                                         // n is a local minimum
    end
    return FindLocalMinimumHelper(A, 1, n)


Function FindLocalMinimumHelper(A, i, j):
    mid ← ⌊(i+j)/2⌋
    if A[mid] < A[mid-1] and A[mid] < A[mid+1] then
        return mid ;                                       // mid is a local minimum
    end
    if A[mid] > A[mid-1] then
        return FindLocalMinimumHelper(A, i, mid-1)
    end
    else
        return FindLocalMinimumHelper(A, mid+1, j)
    end
```

(b) Give an argument of why your algorithm is correct and prove that it has $O(\log n)$ time complexity.

Answer

Let $P(n)$ be the predicate such that $P(n)$ is true if for any array $A[1, \ldots, n]$ of distinct integers, an index $2 \leq j \leq n-1$ has a *local minimum* such that $A[j] < A[j-1]$ and $A[j] < A[j+1]$. We want to show that $\forall n \in \mathbb{N} \; : P(n)$ is true. We can use strong induction to prove this.

base case: $n = 2$, we first check the boundaries of the array that's indices 1 and n. We see that we can have $A[1] < A[2]$, hence $P(2)$ is true. This check establishes that if boundaries have the local minimum we don't continue with the recursion.

base case: $n = 3$. The algorithm first checks if $A[1] < A[2]$ — if so, it returns index 1 as a local minimum. Otherwise, it checks if $A[3] < A[2]$ — if so, it returns index 3 as a local minimum. If neither of these conditions hold, then $A[2] < A[1]$ and $A[2] < A[3]$, which means index 2 is a local minimum, and the recursive helper function will return it upon checking the midpoint, and this establishes that $P(3)$ is true.

inductive case: Fix $k \geq 3$, and assume the inductive hypothesis: for all $3 \leq a \leq k$, every array $A[1, \ldots, a]$ of distinct integers has at least one local minimum, and the algorithm finds it correctly.

Now consider an array $A[1, \ldots, k+1]$. Let $m = \lfloor \frac{i+j}{2} \rfloor$, where $i = 1$ and $j = k+1$. The algorithm first checks whether $A[m]$ is a local minimum:

- If $A[m] < A[m-1]$ and $A[m] < A[m+1]$, then m is a local minimum, and the algorithm returns m.

Otherwise, we must have one of the following:

- **Case 1:** $A[m] > A[m-1]$

  Since all elements are distinct, this implies that $A[m-1] < A[m]$, so the value decreases as we move to the left. Therefore, a local minimum must exist in the subarray $A[i,\ldots,m-1]$. We exclude $A[m]$ from the recursive call since it is not a local minimum. The algorithm recurses on this strictly smaller subarray of atmost size k, and by the inductive hypothesis, it will correctly return a local minimum.

- **Case 2, Else:** $A[m] < A[m-1]$

  This implies that $A[m+1] < A[m]$, so the value decreases as we move to the right. A local minimum must therefore exist in the subarray $A[m+1,\ldots,j]$. Again, we exclude $A[m]$ from the recursive call for the same reason. The inductive hypothesis ensures the recursive call will correctly find a local minimum in this subarray.

These two cases are mutually exclusive when $A[m]$ is not a local minimum. Therefore, the algorithm will always recurse into a valid subarray that contains a local minimum. Thus, it correctly returns a local minimum for input size $k+1$.
By strong induction, $P(n)$ is true for all $n \geq 2$, and the algorithm correctly finds a local minimum in any array $A[1\ldots n]$ of distinct integers.

time complexity: We see that the combine time of the algorithm is $O(1)$ because it is a single comparison, and we pick one half of the array to recurse hence we the following recurrence relation for $T(n)$:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

where $a = 1$ and $b = 2$. We can see that the tree decays by 2^i at each level of the recursion, hence the number of levels of the tree is k such that:

$$\frac{n}{2^k} = 1$$

$$k = \lg n$$

By master theorem we can see that the time complexity is:

$$\begin{aligned}
f(n) &= O(1) \\
n^{\log_b a} &= n^{\log_2 1} = n^0 \quad \text{this fits into case 2 where } k = 0 \\
f(n) &= O(n^{\log_2 2^0} \lg^0 n) = O(1) \\
T(n) &= \Theta(n^{\log_2 1} \lg^{0+1} n) \\
&= \Theta(n^0 \lg n) \\
&= \Theta(\lg n)
\end{aligned}$$

4. **StockPicking**
We solved the Stock Picking problem in class, employing the divide-and-conquer technique to design a linear-time recursive algorithm. Design a simple linear-time algorithm for the same problem without using any recursion.

Answer

Algorithm StockPicking

```
Function StockPicking(A[1...n]):
    min ← A[1]
    bestProfit ← 0
    for i ← 2 to n do
        profit ← A[i] − min
        bestProfit ← max(bestProfit, profit)
        min ← min(min, A[i])
    end
    return bestProfit
```

5. **Sperner**
Given an $n > 1$ and an array $A[1 \dots n]$ of 0s and 1s such that $A[1] \neq A[n]$, a transition index is an index $1 \leq i \leq n-1$ such that $A[i] \neq A[i+1]$ (i.e., either $A[i] = 0$ and $A[i+1] = 1$ or $A[i] = 1$ and $A[i+1] = 0$).

(a) Prove that any such array has at least one transition index.

Answer

Suppose for the sake of contradiction there is no transition index for all arrays $A[1, \dots, n]$ of length n for $n \geq 2$. This means there is no i for $1 \leq i \leq n-1$ such that $A[i] \neq A[i+1]$, and for all i in the array, $A[i] = A[i+1]$. Now pick any erray $A[1, \dots, n]$ of length n. We see that $A[1] = A[2] = A[3] = \dots = A[n]$ hence $A[1] = A[n]$ which is a contradiction. Therefore, we conclude that there exists a transition index i such that $A[i] \neq A[i+1]$ for all arrays $A[1, \dots n]$ of length n where $A[1] \neq A[n]$.

(b) Design an algorithm to compute a transition index of such an array A.

Answer

Algorithm FindTransitionIndex

```
Function FindTransitionIndex(A, i, j):
    if j = i + 1, and A[i] ≠ A[j] then
        return i
    end
    mid ← ⌊(i+j)/2⌋
    if A[mid] = A[i] then
        return FindTransitionIndex(A, mid, j)
    end
    else
        return FindTransitionIndex(A, i, mid)
    end
```

(c) Prove that your algorithm is correct

Answer

Let $P(n)$ be the predicate that states: for any array $A[1, \ldots, n]$ of 0s and 1s such that $A[1] \neq A[n]$, there exists an index $1 \leq i \leq n-1$ such that $A[i] \neq A[i+1]$. We aim to prove $P(n)$ holds for all $n \geq 2$ by strong induction.

Base case: $n = 2$. Since $A[1] \neq A[2]$, a transition occurs at index 1, so $P(2)$ holds.

Inductive step: Fix $k \geq 2$, and assume $P(a)$ holds for all $2 \leq a \leq k$. We wish to show that $P(k+1)$ holds. Let $A[1, \ldots, k+1]$ be any array with $A[1] \neq A[k+1]$, and let $i = 1$, $j = k+1$, and $\text{mid} = \lfloor (i+j)/2 \rfloor$. Since $\text{mid} \leq k$, the inductive hypothesis guarantees a transition exists within any subarray of length $\leq k$ that satisfies the precondition.

The algorithm proceeds by comparing $A[\text{mid}]$ to $A[i]$:

(a) If $A[\text{mid}] = A[i]$, then $A[\text{mid}] \neq A[j]$ by the assumption $A[i] \neq A[j]$. Therefore, a transition must occur in $A[\text{mid}, \ldots, j]$, and the algorithm recurses on that subarray.

(b) If $A[\text{mid}] \neq A[i]$, then a transition exists in $A[i, \ldots, \text{mid}]$, and the algorithm recurses on the left half.

In both cases, the recursive call reduces the problem to a smaller subarray that satisfies the precondition and has length at most k, so the inductive hypothesis applies. Thus, $P(k+1)$ holds.

By the principle of strong induction, $P(n)$ is true for all $n \geq 2$.

(c) Compute the time complexity of your algorithm as a Big-O asymptotic.

Answer

We see that the combine time of the algorithm is $O(1)$ because it is a single comparison, and we pick one half of the array to recurse hence we the following recurrence relation for $T(n)$:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

where $a = 1$ and $b = 2$. We can see that the tree decays by 2^i at each level of the recursion, hence the number of levels of the tree is k such that:

$$\frac{n}{2^k} = 1$$

$$k = \lg n$$

By master theorem we can see that the time complexity is:

$$\begin{aligned}
f(n) &= O(1) \\
n^{\log_b a} &= n^{\log_2 1} = n^0 \quad \text{this fits into case 2 where } k = 0 \\
f(n) &= O(n^{\log_2 2^0} \lg^0 n) = O(1) \\
T(n) &= \Theta(n^{\log_2 1} \lg^{0+1} n) \\
&= \Theta(n^0 \lg n) \\
&= \Theta(\lg n)
\end{aligned}$$