CS3230 Design and Analysis of Algorithms

AY2021/22 Semester 2

1. Introduction

1.1. Adversary Argument

- There are two different cases resulting in different outcomes but cannot be differentiated.

1.2. Minimum Step Problems

- No. of comparisons to find largest element: $n-1$
  - Second largest element: $n-1+\lg n-1$
- No. of comparisons in sorting algorithm: $n \lg n$
- No. of edges checked to tell connectivity: $\binom{n}{2}$

2. Asymptotic Analysis

2.1. Asymptotic Notations

Notation	Definition	$\lim_{n\to\infty} \frac{f(n)}{g(n)}$
$f(n)=O(g(n))$	$\exists c>0, n_0>0$ s.t. $\forall n \ge n_0, 0 \le f(n) \le cg(n)$	$<\infty$
$f(n)=o(g(n))$	$\forall c>0, \exists n_0>0$ s.t. $\forall n \ge n_0, 0 \le f(n) < cg(n)$	$=0$
$f(n)=\Omega(g(n))$	$\exists c>0, n_0>0$ s.t. $\forall n \ge n_0, o \le cg(n) \le f(n)$	>0
$f(n)=\omega(g(n))$	$\forall c>0, \exists n_0>0$ s.t. $\forall n \ge n_0, 0 \le cg(n) < f(n)$	$=\infty$
$f(n)=\Theta(g(n))$	$\exists c_1, c_2>0, n_0>0$ s.t. $\forall n \ge n_0, c_1 g(n) \le f(n) \le c_2 g(n)$	$(0,\infty)$

2.2. Useful Facts

- $\forall k, d>0, (\lg n)^k = o(n^d)$
- $\forall d>0, u>1, n^d = o(u^n)$
- Stirling's Formula: $n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$
  - $\lg(n!) = \Theta(n \lg n)$
  - $\lg \lg n + \lg \lg \frac{n}{2} + \lg \lg \frac{n}{4} + \cdots + 1 = \lg(\lg n!) = \lg n \lg \lg n$
- Harmonic Series: $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} = \Theta(\lg n)$
- Decision Tree with n variables (e.g. sorting):
  - $h = \Omega(\lg(n!)) = \Omega(n \lg n)$
- $\lim_{n\to\infty} \left(\frac{n-1}{n}\right)^n = \frac{1}{e}$

3. Iteration, Recursion and Divide and Conquer

3.1. Correctness of Iterative Algorithm (Loop Invariant)

- Initialisation: The invariant is true before the first iteration of the loop.
- Maintenance: If the invariant is true before an iteration, it remains true before the next iteration.
- Termination: When the algorithm terminates, the invariant provides a useful property for showing correctness.

3.2. Correctness of Recursive Algorithm (Strong Induction)

- Prove base cases.
- Assuming the algorithm works for smaller cases, show that it works correctly.

3.3. Solve a Recurrence

- Recursion Tree: Draw the recursion tree and count total number of operations.
- Master Method for $T(n) = aT\left(\frac{n}{b}\right) + \Theta(f(n))$:

Condition	Solution
$f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$	$T(n) = \Theta(n^{\log_b a})$
$f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \ge 0$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
$f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ $af\left(\frac{n}{b}\right) \le cf(n)$ for some $c < 1$	$T(n) = \Omega(f(n))$

- Substitution Method:
  - Guess the form of the solution.
  - Verify by induction.

Example: Solve $T(n) = 4T\left(\frac{n}{2}\right) + n$.

1. Guess $T(n) = O(n^2)$. Assume $T(1) = q$.
2. We are to show that $\exists c_1, c_2 > 0, n_0 > 0$ s.t. $\forall n \ge n_0, 0 \le T(n) \le c_1 n^2 - c_2 n$.
3. Set $c_1 = q+1, c_2 = 1, n_0 = 1$.
4. Base case: $T(1) = q \le (q+1) - 1$.
5. Recursive case:

$$T(n) = 4T\left(\frac{n}{2}\right) + n \le 4\left(c_1 \cdot \frac{n^2}{4} - c_2 \cdot \frac{n}{2}\right) + n = n^2 - n = c_1 n^2 - c_2 n$$

4. Average Case Analysis and Randomised Algorithms

4.1. MergeSort vs QuickSort

- MergeSort is more efficient theoretically, but QuickSort is preferred empirically.
  - MergeSort requires extra memory.
  - Cache misses.
- Colin McDiarmid Theorem: Probability that the runtime of Randomised QuickSort exceeds average by $x\% = n^{-\frac{x}{100}\ln\ln n}$.

4.2. Geometric Distribution

- Suppose $X \sim Geo(p)$, then $E(X) = \frac{1}{p}$.

5. Hashing

5.1. Universal Hashing

- Suppose $\mathcal{H}$ is a set of hash functions mapping U to $[M]$. We say $\mathcal{H}$ is universal if:

$$\forall x \ne y, \frac{|h \in \mathcal{H} : h(x) = h(y)|}{|\mathcal{H}|} \le \frac{1}{M}$$

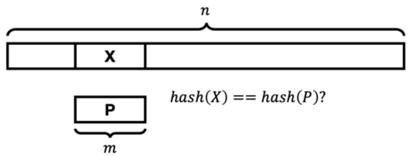
  That is, if we choose h randomly from $\mathcal{H}$, then for any $x \ne y$, the probability of them having the same hash value is smaller than or equal to $\frac{1}{M}$.
- Indicator Variable: $X = \begin{cases} 1, \text{if event } A \text{ occurs} \\ 0, \text{if event } A \text{ does not occur} \end{cases}$
  - For n elements, the expected number of collisions between any pair of them is:

$$\le \binom{n}{2} \cdot \frac{1}{M}$$

5.2. Karp-Rabin Algorithm

- Faster string equality:
  Total runtime for pattern matching

$$= |hash_P| + (n-m+1)(hash_X + O(1))$$



- Rolling hash – Division Hash:

1. Choose p to be a random prime number in the range $\{1, \dots, K\}$.
2. Define, for any integer x, $h_p(x) = x \bmod p$.

  - Useful fact: Number of prime numbers in $\{1, \dots, K\} > K / \ln K$.
  - Hence, if $0 \le x < y < 2^b$, then

$$\Pr(h_p(x) == h_p(y)) < \frac{b \ln K}{K}$$

  - Set $K = 200mn \ln(200mn)$. Then the probability of getting a false positive is $<$ 1%.
  - Roll from $T[1, \dots, m]$ to $T[2, \dots, m+1]$:

$$h_p(X') = \left(h_p(X) - T[1] \cdot h_p(2^{m-1})\right) \cdot 2 + T[m+1] \pmod p$$

- Monte-Carlo Algorithm

1. Pick random prime p from $\{1, \lceil 200mn \ln 200mn \rceil\}$.
2. Compute $h_p(P)$, $h_p(2^m)$ and $h_p(T[1, \dots, m])$.
3. Check if $h_p(P) == h_p(T]1, \dots, m])$.
4. Start rolling and check each substring equality.

  - Runtime: $O(m+n)$
  - Error probability: $< 1\%$

6. Amortized Analysis

6.1. Aggregate Method

- Average cost of n operations:

$$\frac{\sum_{i=1}^{n} t(i)}{n}$$

6.2. Accounting Method

- Basic idea: Save additional money for fast method, use the saved money for costly method.

6.3. Potential Method

$$c_i = t_i + \phi(i) - \phi(i-1)$$

- c_i: Amortised cost of i-th operation
- t_i: True cost of i-th operation
- ϕ: Potential function associated with the algorithm/data structure
- $\phi(i)$: Potential at the end of i-th operation
- $c_i = t_i + \phi(i) - \phi(i-1)$

7. Dynamic Programming

7.1. Knapsack Problem

Given W, the total weight that a knapsack can hold, and a set of items (w_i, v_i) where $i = 1, \dots, n$ with weight w_i and value v_i, what is the optimal strategy to get the highest value?

1. Initialise a table m of size $n \times W$.
2. **for** $i = 1, \dots, n$ **do**
   for $j = 0, \dots, W$ **do**
     if $j \geq W[i]$ **then**
       $m[i,j] \leftarrow \max(m[i-1, j - W[i]] + v[i], m[i-1, j])$
     else
       $m[i,j] = m[i-1, j]$
3. **Return** $m[n, W]$.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	2	2	10	12	12	12	12	12	12	12	12	12	12	12
4	0	2	3	3	10	12	13	13	13	13	13	13	13	13	13	13
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	**15**

8. Greedy Algorithm

8.1. Correctness of Greedy Algorithm

Optimal Substructure

1. Suppose S is any optimal solution, and S contains item i.
2. Claim: $S - \{i\}$ is optimal for the subproblem with i removed and *necessary changes made* (e.g. n to $n-1$).
3. Cut & Paste Proof: Assume instead T is the optimal solution to the subproblem, then $T + \{i\}$ would be optimal for the current problem, leading to contradiction.

Greedy-Choice Property

1. Suppose i is the element that *is chosen greedily* (e.g. max).
2. Claim: There exists an optimal solution that contains i.
3. Proof: Suppose there is an optimal solution that does not contain i. By replacing any item in the solution with i, the solution will become *more or as optimal* (elaboration), leading to contradiction.

Conclusion

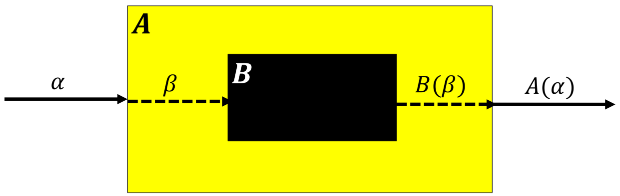
By Greedy-Choice Property, the *greedily chosen* (elaboration) element is in the optimal solution. By Optimal Substructure, this can be combined with solutions to remaining subproblems.

9. Reduction and Intractability

9.1. Reduction

- Polynomial-time Reduction: $A \leq_P B$ if there exists a $p(n)$ time reduction from A to B where $p(n) = O(n^c)$ for some constant c.
- Correctness of Reduction

1. Reduction runs in polynomial time.
2. If α is a YES-instance of A, then β is a YES-instance of B.
3. If β is a YES-instance of B, then α is a YES-instance of A.



9.2. NP-Completeness

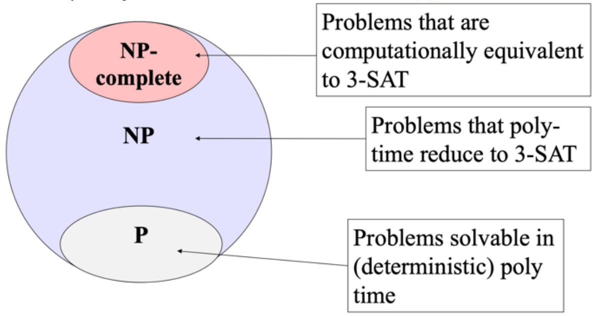
Proof of NP

A YES-instance has a certificate that can be verified in polynomial time.

Proof of NP-hard

To show that a valid polynomial time reduction exists from another NP-hard problem A.
1. The reduction should run in polynomial time.
2. If the instance of the current problem X is a YES-instance, then the corresponding instance of A is also a YES-instance.
3. If the instance of A is a YES-instance, then the corresponding instance of X is also a YES instance.

9.3. Complexity Classes



Good luck!

Additional Thoughts

This module is quite with intuition.