COMP 355
Advanced Algorithms

Algorithm Design Review:
Mathematical Background

Rhodes College

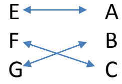
Stable Marriage

The Gale-Shapley Algorithm

```
// Input: 2n preference lists, each consisting of n names.
// Output: A matching that pairs each man with each woman.
Initially all men and all women are unengaged
while (there is an unengaged man who hasn't yet proposed to every woman) {
    Let m be any such man
    Let w be the highest woman on his list to whom he has not yet proposed
    if (w is unengaged) then she accepts ((m, w) are now engaged)
    else {
        Let m' be the man w is engaged to currently
        if (w prefers m to m') {
            Break off the engagement (m', w)
            Create the new engagement (m, w) (upgrade)
            Man m' is now unengaged
        }
    }
}
```

Men		
Eddy (E)	Freddy (F)	Gerry (G)
B	B	C
A	C	B
C	A	A

Women		
Anny (A)	Betty (B)	Carry (C)
G	G	E
F	E	F
E	F	G



Polynomial Running Time

- Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
  - Typically takes 2^N time or worse for inputs of size N.
  - Unacceptable in practice.

n! for stable matching with n men and n women

- Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor **c**.

> There exists constants c > 0 and d > 0 such that on every input of size n, its running time is bounded by $c\,n^d$ steps.

- Def. An algorithm is poly-time if the above scaling property holds.

Worst-Case Analysis

- Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.
- Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.

Rhodes College

Worst-Case Polynomial Time

An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

Rhodes College

Big-O Notation

- Asymptotic O-notation ("big-O") provides a way to simplify the messy functions that often arise in analyzing the running times of algorithms
- Allows us to ignore less important elements (constants)
- Focus on important issues (growth rate for large values of n)

$$\begin{aligned}
f_1(n) &= 43n^2\log^4 n + \underline{12n^3\log n} + 52n\log n && \in O(n^3\log n)\\
f_2(n) &= \underline{15n^2} + 7n\log^3 n && \in O(n^2)\\
f_3(n) &= 3n + 4\log_5 n + \underline{91n^2} && \in O(n^2).
\end{aligned}$$

Formal Definition Big-O

- Formally, *f(n)* is $O(g(n))$ if there exist constants c > 0 and $n_0 \geq 0$ such that, $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$.
- Thus, big-O notation can be thought of as a way of expressing a sort of fuzzy "≤" relation between functions, where by fuzzy, we mean that constant factors are ignored and we are only interested in what happens as *n* tends to infinity.

Intuitive Form of Big-O

$$f(n)\ is\ O\big(g(n)\big)\ if\ \lim_{n\to\infty}\frac{f(n)}{g(n)} \geq c, for\ some\ constant\ c\ \geq 0.$$

For example, if $f(n) = 15n^2 + 7n\log^3 n$ and $g(n) = n^2$, we have f(n) is O(g(n)) because

$$\begin{aligned}\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\left(\frac{15n^2+7n\log^3 n}{n^2}\right) = \lim_{n\to\infty}\left(\frac{15n^2}{n^2}+\frac{7n\log^3 n}{n^2}\right)\\ &= \lim_{n\to\infty}\left(15+\frac{7\log^3 n}{n}\right) = 15.\end{aligned}$$

In the last step of the derivation, we have used the important fact that log n raised to any positive power grows asymptotically more slowly than n raised to any positive power.

Useful facts about limits

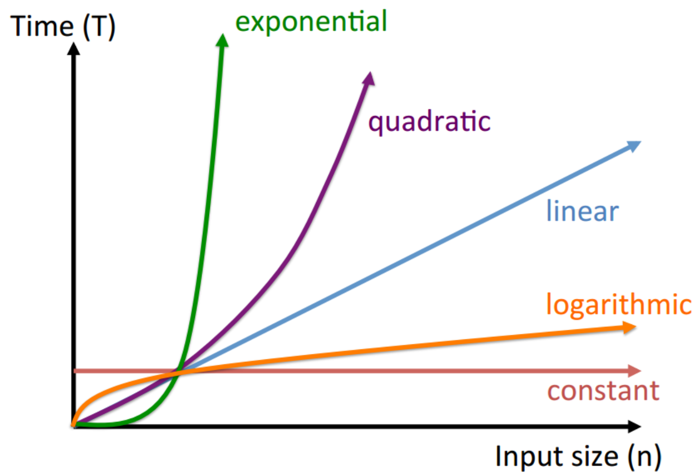
- For $a, b > 0$, $\lim_{n\to\infty} \frac{(\log n)^a}{n^b} = 0$ (polynomials grow faster than polylogs).
- For $a > 0$ and $b > 1$, $\lim_{n\to\infty} \frac{n^a}{b^n} = 0$ (exponentials grow faster than polynomials).
- For $a, b > 1$, $\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = c \neq 0$ (logarithm bases do not matter).
- For $1 < a < b$, $\lim_{n\to\infty} \frac{a^n}{b^n} = 0$ (exponent bases do matter).

Rhodes College

Survey of Common Running Times

- Linear Time: O(n)
- Linearithmic Time: O(n log n)
- Quadratic Time: $O(n^2)$
- Cubic Time: $O(n^3)$
- Polynomial Time: $O(n^k)$
- Exponential Time: $O(2^{n^k})$

Comparison of Running Times



Other Asymptotic Forms

- Big-O has a number of relatives, which are useful for expressing other sorts of relations.
- More on these next time!

Rhodes College

Summations

- Naturally arise in analysis of iterative algorithms
- More complex forms of analysis, such as recurrences, are often solved by reducing to summations
- Solving a summation means reducing it to a closed-form formula
  - No summations, recurrences, integrals, or other complex operators
- Often don't need to solve a summation exactly to find the asymptotic approximation

Rhodes College

Summations With General Bounds

Summations with general bounds: When a summation does not start at the 1 or 0, as most of the above formulas assume, you can just split it up into the difference of two summations. For example, for $1 \leq a \leq b$

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i).$$

Rhodes College

Linearity of Summation

Linearity of Summation: Constant factors and added terms can be split out to make summations simpler.

$$\sum(4+3i(i-2)) \;=\; \sum 4+3i^2-6i \;=\; \sum 4+3\sum i^2-6\sum i.$$

Apply the formulas to each summation individually.

Approximate Using Integrals

Approximate using integrals: Integration and summation are closely related. (Integration is in some sense a continuous form of summation.) Here is a handy formula. Let $f(x)$ be any *monotonically increasing function* (the function increases as x increases).

$$\int_0^n f(x)dx \;\leq\; \sum_{i=1}^{n} f(i) \;\leq\; \int_1^{n+1} f(x)dx.$$

Example: Previous Larger Element

Given a sequence of numeric values, <$a_1, a_2, \ldots, a_n$>. For each element a_i, for $1 \le i \le n$, we want to know the index of the rightmost element of the sequence <$a_1, a_2, \ldots, a_{i-1}$> whose value is strictly larger than a_i. If no element of this subsequence is larger than a_i then, by convention, the index will be 0. (Or, if you like, you may imagine that there is a fictitious sentinel value $a_0 = \infty$.) More formally, for $1 \le i \le n$, define p_i to be
$p_i = \max\{j \mid 0 \le j < i \text{ and } a_j > a_i\}$, where $a_0 = \infty$ (see Fig. 2).

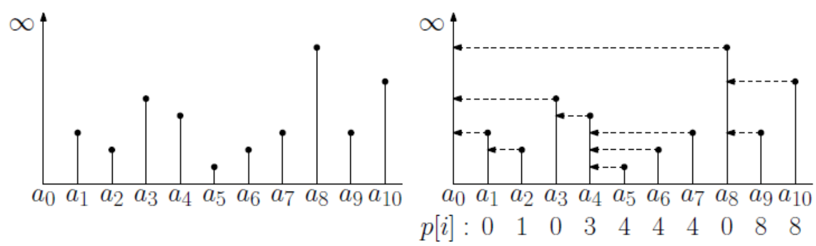


Fig. 2: Example of the previous larger element problem.

Naive Algorithm For Previous Larger Element

Previous Larger Element (Naive Solution)

```
// Input: An array of numeric values a[1..n]
// Returns: An array p[1..n] where p[i] contains the index of the previous
//   larger element to a[i], or 0 if no such element exists.
previousLarger(a[1..n]) {
    for (i = 1 to n)
        j = i-1;
        while (j > 0 and a[j] <= a[i]) j--;
        p[i] = j;
    }
    return p
}
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 = 1 + 2 + \ldots + (n-2) + (n-1) = \sum_{i=1}^{n-1} i.$$

$$T(n) = \frac{(n-1)n}{2}.$$

Rhodes College

Recurrences

Arise naturally in analysis of divide-and-conquer algorithms

- **Divide**: Divide the problem into two or more sub-problems (ideally of roughly equal sizes)
- **Conquer**: Solve each sub-problem recursively
- **Combine**: Combine the solutions to the sub-problems into a single global solution.

Recurrences

- To analyze recursive procedures such as divide-and-conquer, we need to set up a recurrence.
- Example: Suppose we break a problem into two sub-problems, each of size roughly n/2.
- Additional overhead of splitting and merging the solutions is O(n).
- When sub-problems are reduced to size 1, we can solve them in O(1) time.
- Ignoring constants and writing O(n) as n, we get:

$$T(n) = 1 \text{ if } n = 1,$$
$$T(n) = 2T(n/2) + n \text{ if } n > 1$$

Example Problem

- Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2, & if\ n = 2, \\ 2T\left(\frac{n}{2}\right) + n, & if\ n = 2^k, for\ k > 1 \end{cases}$$

is $T(n) = n \lg n$

Next Time

- Other Asymptotic Forms
- Read Section 2.2