CSCI 3104 Notes

William Farmer

April 6, 2014

Contents

1 Complexity

1.1 Atomic Operations

Any fundamental operation that is performed: $+, *, /, -, <<, >>, <, >, =$, assignment.

1.2 Big-O Notation

Big-O is a method to asymptotically analyze an equation or algorithm. When we analyze a process we're concerned about the complexity of two things: time and space.

These could be represented by two different equations

$$\begin{cases} Time \to f(n) \\ Space \to g(n) \end{cases}$$

and further summarized by a limit of the ratio[1]

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Theorem 1. *Let $f(n)$ and $g(n)$ be functions $\mathbb{N} \to \mathbb{R}$. We say $f = O(g)$, which means if there is $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

1.3 Big-Θ Notation

Big-Θ notation is the original notation and all others are merely spin-offs.

Theorem 2. *A function (or algorithm), $f(n)$, is Big-Θ of $g(n)$ if $(\exists c_1 > 0, c_2 > 0, \forall n > n_0 > 0)[c_1 g(n) \leq f(n) \leq c_2 g(n)]$. We denote this $f(n) = \Theta(g(n))$.*

2 Sorting

2.1 Divide and Conquer

This is a classic strategy to solve complex problems.

1. Divide the problem into n sub-problems
2. Conquer each sub-problem individually
3. Combine the sub-problems back into the original problem.

A typical divide and conquet algorithm will look as such.

```
1    def fun(n)
2        if n == trivial:
3            solve and return
4        else
5            partA = fun(n_)
6            partB = fun(n - n_)
7            AB    = combine(A, B)
8            return AB
```

[1] This ratio can be reduced by L'Hospital's Rule until we determine the constant ratio

This code will create a binary tree with depth n containing trivial problems on the left branches. This is commonly known as *tail recursion*. The problem is being divided, but very slowly - removing a small part of the problem every step.

2.1.1 Recurrence Relations

Generally of the form

$$T(n) = \underbrace{a}_{\text{Number of Recursive Calls}} \cdot \overbrace{T(\underbrace{g(n)}_{\text{Method of Reducing the Problem}})}^{\text{Cost of Recursion}} + \underbrace{f(n)}_{\text{Cost of Non-Recursive Work}}$$

If the problem is divided in twain it will generate a binary tree with approximate depth of $\log_2(n)$.

2.2 Bubblesort

While a pair of entries is out of order, loop and fix each pair each step.

2.3 Quicksort

Pick a pivot and sort the items in relation to the pivot.

```
1  # precon:  A is an array to be sorted, p >= 1, r <= |A|
2  # postcon: A[p:r] is sorted
3  def quicksort(A, p, r):
4      if p < r:
5          q = partition(A, p, r)
6          quicksort(A, p, q - 1)
7          quicksort(A, q - 1, r)
8
9  # precon:  A[p:r] is input, p >= 1, r <= |A|
10 #          A[r] is the pivot
11 # postcon: A_ is A after function
12 #          A_[p:r] contains some elements in A[p:r]
13 #          A_[p:res - 1] <= A[r], A_[res] - A[r]
14 #          A_[res + 1:r] > A[r]
15 def partition(A, p, r):
16     x = A[r]                        # Choose Pivot
17     i = p - 1
18     for (j=p; j <= r - 1; j++):
19         if A[j] <= x:
20             i++
21             exchange(A[i], A[j])
22     exchange(A[i + 1], A[r])
23     return i + 1
```

2.3.1 Loop Invariants

These are claims that hold at the beginning of each loop.

1. If $p \leq k \leq i$, then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$ then $A[k] > x$
3. If $k = r$ then $A[k] = x$

Verify that $\boxed{1}$ holds at the beginning of the loop.
Verify that if it holds at the $(i-1)^{th}$ iteration, then it holds at the i^{th} iteration.
Verify that if it holds when the loop terminates, then $A[p, r]$ is partitioned.

2.3.2 Running Time

The worst case for this algorithm is when we have a very unbalanced tree.

$$T(n) = T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$$

Therefore, the best case occurs when we have a perfectly balanced binary tree.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta()$$

2.3.3 Randomizing

As you can see, there is a very distinct case where the list results in an $O(n^2)$ algorithm. We solve this by randomizing the input list.

The average case of randomized quicksort is the same as quicksort's average case.

```
1  def random_qs(A, p, r):
2      q = random_partition(A, p, r)
3      random_qs(A, p, q - 1)
4      random_qs(A, q + 1, r)
5
6  def random_partition(A, p, r)
7      i = random.int(p, r)  # Use a random pivot
8      swap(A[i], A[r])
9      x = A[r]
10     i = p - 1
11     for (j=p; j <= r - 1; j++):
12         if A[j] <= x:
13             i += 1
14             exchange(A[i], A[j])
15     exchange(A[i + 1], A[r])
16     return i + 1
```

3 Greedy Algorithms

This algorithms arise from cases where we wish to select the largest continuous time, area, space, etc. so that the regions don't overlap.[2] We can achieve this with greedy algorithms.

[2] Also known as the interval scheduling problem

This strategy works by selecting the largest items first, and then selecting by minumum conflict.

1. Order all intervals by end time. $O(n \log(n))$
2. Check each interval based on the overlap. $O(n)$

The formal problem is stated as such: We have a set of n intervals, $\{1, 2, 3, \cdots, n\}$, and we have a start and finish time. We call a subset of intervals compatible if they don't overlap.

```
# Input: intervals
# Output: Maximal set of compatible intervals
# Let I = all intervals
#     O = empty list
def greedy_by_finish_time(I):
    while I is not empty:
        choose i from I with smallest finish time f(i)
        add to O
        delete all intervals from I that are not compatible with i
    return O
```

4 Recurrence Relations

4.1 Characteristic Polynomial

4.2 Unrolling

Substitute the next equation in the current.

4.3 Master Theorem

There are three cases to consider.

$$T(n) = a \cdot T(n/b) + f(n)$$

$$\begin{cases} f(n) = O\left(n^{\log_b(a-\epsilon)}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b(a)}\right) \\ f(n) = O\left(n^{\log_b(a)}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b(a)} \cdot \log(a)\right) \\ \left.\begin{array}{l} f(n) = \Omega\left(n^{\log_b(a+\epsilon)}\right) \\ a \cdot f(n/b) \leq c \cdot f(n) \end{array}\right\} \Rightarrow T(n) = \Theta(f(n)) \end{cases}$$

5 Basic Probability

The expectation of a discrete event, $E(x)$ is defined as

$$E(x) = \sum_x xPr(X = x)$$

For continuous probabilities, we use

$$E(x) = \int_x xPr(x)\, dx$$

If there are two random variables, in order to identify the probability of both events occurring can be found by multiplication if and only if they are mutually exclusive.

$$Pr(X = x, Y = y) = Pr(X = x) \cdot Pr(Y = y)$$

We can also use indicator random variables to count how many events occur.

$$I(A) = \begin{cases} 1 \text{ if event } A \text{ occurs} \\ 0 \text{ otherwise} \end{cases}$$

Multiple random variables can be combined using addition.

$$E\left(\sum_i X_i\right) = \sum_i E(X_i)$$

6 Knapsack Problem

6.1 Easy Version

We have some capacity k that limits our endeavours. We then have a set of items with two properties, their values, and their weight.

$$\begin{cases} \text{Value: } \{v_1, v_2, \ldots, v_n\} = \sum_{i=1}^{n} f_i \cdot v_i \\ \text{Weight: } \{w_1, w_2, \ldots, w_n\} = \sum_{i=1}^{n} f_i \cdot w_i \leq k \end{cases}$$

We also have a fraction $f_i \in [0, 1]$ that represents the fraction of the items we take. We wish to maximize the total value of items that we take, while satisfying the total weight.

To solve this problem, we should look at the items' value density, i.e. the items with the least weight, but maximum value, $\frac{v_n}{w_n}$. In other words, take the greedy approach, that is to take the best items until we can only take a fractional item, in which case we stop. This greedy approach is optimal, and takes $O(n \log(n))$ time to complete.

6.2 Hard Version (0-1 Knapsack Problem)

In the hard version, our fraction adjusts to $f_i \in \{0, 1\}$. This essentially means that we are no longer allowed to take fractional items, are are limited to boolean choices. This runs in $O(n \cdot k)$ time using a dynamic programming solution.

For our solution, we need to use dynamic programming to construct a table.

$$v_{0,0} \quad v_{0,1} \quad \cdots \quad v_{0,w-1} \quad v_{0,w}$$

Initially we set $v[0, w] = 0$ where $0 \leq w \leq k$, followed by setting $v[i, w] = -\infty$ where $w < 0$, $v[i, w] \equiv$ the maximum value of items $[1, i]$ with combined weight at most w. Now we can start establishing entries in our table. For each item in $v[i, w]$ we have two options.
1. Take i if $w_i \leq w$ also if the optimal solution $v[i, w]$ includes i. Leaving it gives the optimal solution $v[i - 1, w - w_i]$.
2. Leave i if the optimal solution for items $\{1, \ldots, i - 1\}$ with weight w is $v[i - 1, w]$.

These two rules can be rewritten, giving us

$$v[i,w] = \max(v[i-1,w], v_i + v[i-1, w-w_i]) \text{ for } \begin{cases} 1 \leq i \leq n \\ 0 \leq w \leq k \end{cases}$$

7 Dynamic Programming

In dynamic programming we have some semblance of a subproblem structure, where as we solve each subproblem it helps us solve the next sub-problem. In other words, we remember the solutions to previous subproblems and thereby build solutions to the next problems off of what we remember[3] from previous problems. In almost every dynamic programming solution we build a table of solutions.[4]

[3] Memoization
[4] See PS3, Dumbledore's Algorithm for an example of Memoization