COMP20007 Design of Algorithms

Week 3 Workshop Solutions

Tutorial

0. Sums

(a)

$$\sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \ldots + 1}_{n \text{ times}} = n$$

(b)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \quad \text{(Triangle numbers formula)}$$

(c)

$$\sum_{i=1}^{n}(2i+3) = 2\sum_{i=1}^{n} i + \sum_{i=1}^{n} 3 = 2 \times \frac{n(n+1)}{2} + 3 \times n = n^2 + 4n$$

(d)

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i} 1 = \sum_{i=0}^{n-1}(i+1) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

(e)

$$\sum_{i=1}^{n}\sum_{j=1}^{m} ij = \left(\sum_{i=1}^{n} i\right)\left(\sum_{j=1}^{m} j\right) = \left(\frac{n(n+1)}{2}\right)\left(\frac{m(m+1)}{2}\right) = \frac{nm(n+1)(m+1)}{4}$$

(f)

$$\sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1 - x} \quad \text{(Geometric series)}$$

1. Complexity classes

(a) $\frac{1}{2}n^2 \in \Omega(3n)$ – ignore constants, n^2 grows faster than n

(b) $n^2 + n \in \Theta(3n^2 + \log n)$ – ignore constants, the fastest growing terms are both n^2

(c) $n \log n \in O\left(\frac{n}{4}\sqrt{n}\right)$ – ignoring constants the difference here is $\log n$ vs. $\sqrt{n}$

(d) $\log(10n) \in \Theta(\log(n^2))$ – using log laws $f(n) = \log 10 + \log n$ and $g(n) = 2 \log n$

(e) $(\log n)^2 \in \Omega(\log(n^2))$ – we can see that $(\log n)^2/(2 \log n) = \frac{1}{2} \log n$ so $f(n)$ grows faster

(f) $\log_{10} n \in \Theta(\ln n)$ – using change of base formula $\log_{10} n = \frac{1}{\ln 10} \ln n$

(g) $2^n \in O(3^n)$ – unlike logarithms, the base in the exponential changes the growth rate

(h) $n! \in O(n^n)$ – $1 \times 2 \times \cdots \times n \in O(n \times n \times \cdots \times n)$ but not vice versa

2. Sequential search

(a) general (*i.e.*, all possible inputs): the best case for sequential search is when the element we're searching for is at index 0. So $C_{\text{best}}(n) = 1$. Also $C_{\text{worst}}(n) = n$ occurs when the element is not in the array. So $C_{\text{best}}(n) \leq C(n) \leq C_{\text{worst}} \implies C(n) \in \Omega(1)$ and $C(n) \in O(n)$.

(b) the best case: in the best case $C_{\text{best}}(n) = 1$ so $C_{\text{best}} \in \Theta(1)$.

(c) the worst case: in the words case $C_{\text{worst}}(n) = n$ so $C_{\text{worst}} \in \Theta(n)$.

(d) the average case: let the probability of the element being in the array be p. If the element is not in the array the cost is n. If the element is in the array we assume it's equally likely to be in any of the n indices. Taking the average of the number of comparisons when the element is in each index:

$$\frac{1}{n}(1 + 2 + \cdots + n) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

So the cost is $\frac{n+1}{2}$ with probability p, or n with probability $(1-p)$, *i.e.*,

$$C_{\text{average}}(n) = p \times \frac{n+1}{2} + (1-p) \times n.$$

Since p is a constant here, $C_{\text{average}} \in \Theta(n)$.

4. k-Merge Merging two lists of sizes a and b takes about $a + b$ steps. Merging the first two lists (sizes n and n) will take $2n$ steps. Merging this list with the next list (sizes $2n$ and n) will take $3n$ steps. The next will be $4n$ steps, and so on, until the last list of n is merged with the rest of the $(k-1)n$ items, taking kn steps. In total,

$$2n + 3n + 4n + \cdots + kn = n(2 + 3 + 4 + \cdots + k)$$

Simplifying by recognising the triangle numbers, we're looking at $\Theta(k^2 n)$:

$$n(2 + 3 + 4 + \cdots + k) = n(1 + 2 + 3 + 4 + \cdots + k) - n = n\frac{k(k+1)}{2} - n \in \Theta(k^2 n)$$

A faster algorithm would use the merging strategy from mergesort, merging the lists in pairs. First, merge $k/2$ pairs of length n lists, resulting in $k/2$ lists of length $2n$ (and taking $k/2 \times 2n = kn$ steps). Next, merge $k/4$ pairs of length $2n$ lists, resulting in $k/4$ lists of length $4n$ (and taking $k/4 \times 4n = kn$ steps). Continue, a total of $\log k$ times, and you will have one list of length kn in $\Theta(kn \log k)$ time.

5. Mergesort complexity (optional) Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

We can use three intuitive observations:

- Sorting each half of the input will take the time that mergesort takes for an input of half the size.
- We can merge two sorted lists of size n and m in O(n + m) time.
- A single item is already sorted.

We will formalise the intuition next week, our first insight is that we halve the input size and operate on both halves. Intuitively this leads to the sum of every split performing the original n operations during the merging step. The final intuitive component is how many splits we need to perform. This is governed by the observation that a single item is already sorted and we begin with n elements. The original number of elements, n, can only be halved $\lceil \log_2 n \rceil$ times before we reach a single element.

Then, putting our two insights together we can derive a complexity of $O(n \log n)$.