Solutions to Tutorial 1

CS 213: Data Structures and Algorithms

---

1. What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?

   Solution. Let m and n be the two numbers. It is easy to see that the binary representation of any positive integer k has size $\lfloor \log_2 k \rfloor + 1$, and its unary representation has size k.

   Note: In the unary numeral system, a positive integer k is represented by a string of k 1's.

   - Binary addition: The addition of one bit to another, with a carry value, is a constant $\mathcal{O}(1)$ time operation, since there are only 8 possible cases. Binary addition of m and n involves $\max(\lfloor \log_2 m \rfloor + 1, \lfloor \log_2 n \rfloor + 1)$ such constant-time addition operations. Therefore, the time complexity of binary addition is $\mathcal{O}(\max(\log m, \log n)) = \mathcal{O}(\log m + \log n)$.
   - Binary multiplication: This consists of two parts; multiplying bits and adding bits. Just like addition, the multiplication of a pair of bits is also a constant $\mathcal{O}(1)$ time operation. Each bit of m is multiplied with every bit of n, so there would be a total of $(\lfloor \log_2 m \rfloor + 1)(\lfloor \log_2 n \rfloor + 1)$ bit multiplications. There would also be an equal number of bit additions, since each of the multiplications produces a bit which is then added in its appropriate column. Thus, binary multiplication has a time complexity of $\mathcal{O}(\log m \cdot \log n)$.
   - Unary addition: The addition of two numbers m and n in unary form is essentially their concatenation, since the result would simply be a string of $(m + n)$ 1's. Thus, it would involve iterating over and copying all the digits of both m and n, which brings the time complexity of unary addition to $\mathcal{O}(m + n)$.

---

2. If $f(n) = \mathcal{O}(F(n))$ and $g(n) = \mathcal{O}(G(n))$, show that $h(n) = \dfrac{f(n)}{G(n)} = \mathcal{O}\left(\dfrac{F(n)}{g(n)}\right)$.

   Solution. We have:

   $$f(n) = \mathcal{O}(F(n)) \implies \exists c_1, n_1 \text{ such that } \forall n \geq n_1,\ 0 \leq f(n) \leq c_1 \cdot F(n)$$
   $$g(n) = \mathcal{O}(G(n)) \implies \exists c_2, n_2 \text{ such that } \forall n \geq n_2,\ 0 \leq g(n) \leq c_2 \cdot G(n)$$

   Thus:

   $$\forall n \geq \max(n_1, n_2),\ 0 \leq \frac{f(n)}{G(n)} \leq c_1 c_2 \cdot \frac{F(n)}{g(n)}.$$

Thus, for $c = c_1c_2, n_0 = \max(n_1, n_2)$, we have $\exists c, n_0$ such that $\forall n \geq n_0, 0 \leq \frac{f(n)}{G(n)} \leq c \cdot \frac{F(n)}{g(n)}$

That is, $\frac{f(n)}{G(n)} = \mathcal{O}\left(\frac{F(n)}{g(n)}\right)$.

---

3. Let $f(n)$ and $g(n)$ be asymptotically non-negative. Prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Solution. Since $f(n)$ and $g(n)$ are asymptotically non-negative, we have, for large enough n:

$$f(n) \leq f(n) + g(n),\ g(n) \leq f(n) + g(n) \implies \max(f(n), g(n)) \leq f(n) + g(n)$$

and:

$$f(n) \leq \max(f(n), g(n)),\ g(n) \leq \max(f(n), g(n)) \implies f(n) + g(n) \leq 2 \cdot \max(f(n), g(n))$$

Thus, for large n:

$$\frac{f(n) + g(n)}{2} \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

Therefore, we conclude $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

---

4. Is $2^{n+1} = \mathcal{O}(2^n)$? Is $2^{2n} = \mathcal{O}(2^n)$?

Solution.

- $2^{n+1} = \mathcal{O}(2^n)$ is clearly true, because for all $n \geq 0$, we have $0 \leq 2^{n+1} \leq 2 \cdot 2^n$.
- Suppose there exist c and n_0 such that:

  $$\forall n \geq n_0,\ 0 \leq 2^{2n} \leq c \cdot 2^n \implies \forall n \geq n_0,\ 2^n \leq c.$$

  This is a contradiction, since we also know $\forall n \geq \log_2 c,\ 2^n \geq c$. Thus, there is no such c and n_0 pair. Therefore, $2^{2n} \neq \mathcal{O}(2^n)$.

---

5. There is a stack of n dosas on a tava, all having distinct radii. We want to serve them in order of increasing radii. Only two operations are allowed:

   (a) serve the top dosa,

   (b) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack, flip it upside-down, and put it back.

   Design a data structure to represent the tava, to input a given tava, and to produce an output in sorted order. What is the time complexity of your algorithm?

Solution. Remember the array implementation of a stack? That's exactly how we'll represent our stack of dosas on the tava.

- Initialization and input: Initialize an array `S` of size `n`, to represent the stack. Using the given input, fill the array elements with the radii of dosas on the tava in the initial stack order, with the bottom dosa at index 0 and the top at `n-1`. Maintain a variable `sz` which contains the current size of the stack, and initialize it to `n`.
- Serving the top dosa: This method is essentially carrying out the "pop" operation on our stack `S`. Return the top dosa in the stack, `S[sz-1]`, and decrement `sz` by 1. Clearly, this method takes constant $\mathcal{O}(1)$ time for every call.
- Flipping the top `k` dosas: This method is equivalent to reversing the slice of the array `S` from index `sz-k` to `sz-1`. This is simple enough; initialize an index `i` to `sz-k` and another index `j` to `sz-1`, swap elements at indices `i` and `j`, increment `i` and decrement `j` by 1, and repeat the swap and increment/decrement while `j>i`. This method takes $\mathcal{O}(k)$ time for every call.

Now that our implementation of the dosa stack is ready, we move on to design an algorithm to serve the dosas in increasing order of their radii, with all the changes to the stack being made through one of the two operations implemented above.

We use an idea similar to selection sort. While the stack is not empty, repeat the following:

- Iterate over the array `S` and find the index of the minimum element in the array. Let this index be `m`. This is the position of the dosa having the smallest radius.
- Flip over the top `sz-m` dosas in the stack, using the flipping operation. This brings the smallest dosa from index `m` to index `sz-1`, i.e., the top of the stack.
- Serve the top dosa, using the serving operation. This pops the smallest dosa off the stack.

It's easy to see why this works; we keep serving the minimum size in the remaining stack, so they come out in sorted order.

In a stack of size z, finding the index of the minimum element takes $\mathcal{O}(z)$ time, flipping over the top $k \leq z$ elements takes $\mathcal{O}(k)$ time (and thus $\mathcal{O}(z)$ time), and serving off the top dosa takes $\mathcal{O}(1)$ time. Thus, serving the smallest dosa takes $\mathcal{O}(z)$ time overall, where z is the size of the stack.

This has to be repeated for all dosas, i.e., for stack sizes of $z = n, n-1, \ldots, 1$. Therefore, the algorithm has a time complexity of $\mathcal{O}(n^2)$.

---

6. One problem is to check whether a string of opening and closing brackets of various types is a valid bracket sequence. For example `{(){}[[]]}` and `([[[]]]{})` are valid, while `(){(` and `(()))` are invalid. Write a stack based algorithm to check whether a bracket string is valid.

   Solution. This problem can be solved using a stack. The algorithm proceeds as follows:

   - Initialize an empty character stack `S`.
   - Starting at the first bracket character, iterate over the string of brackets.
     - If the character encountered is an opening bracket (`(`, `{`, or `[`), then push it onto `S`.
     - If the character encountered is a closing bracket (`)`, `}`, or `]`), then pop a character from `S`, if possible. If `S` is non-empty before this (that is, popping an element is possible), and if the popped character is the corresponding opening bracket, then continue the traversal; otherwise, the bracket string is invalid. Note that invalidity of the sequence may be caused here by either an empty stack (no popping possible) or a bracket type mismatch.

- At the end of the traversal, if `S` is empty, then the bracket string is valid; otherwise, it is invalid.

Since character comparison, pushing onto a stack, and popping off the top of a stack are all constant time operations, this algorithm has $\mathcal{O}(n)$ time complexity, where n is the length of the bracket string.

---

7. *The mess table queue.* There is a common mess for K hostels. Each hostel has some $N_1, \ldots, N_K$ students. These students line up to pick up their trays in the common mess. However, the queue is implemented as follows: If a student sees a person from his/her hostel, she/he joins the queue behind this person. This is the "enqueue" operation. The "dequeue" operation is as usual, at the front. Think about how you would implement such a queue. What would be the time complexity of enqueue and dequeue? Do you think the average waiting time in this queue would be higher or lower than a normal queue? Would there be any difference in any statistic? If so, what?

   Solution. First, note that such a queue would consist of at most K *sub-queues*, each one containing students from the same hostel. New students who show up to join the queue get attached to the end of one of these hostel sub-queues.

   We implement this with the following:

   - A global linked list of all students currently in the queue (each student is a node in the list). This linked list is initially empty.
   - Pointers `head` and `tail`, to the front and end of the queue respectively. Both are initially null.
   - An array of pointers `sqend` of size K. `sqend[i]`, for $0 \leq i \leq K-1$, has a pointer to the last student in the sub-queue of hostel $i+1$, if that sub-queue currently exists in the queue; otherwise, `sqend[i]` is null. (The hostel number is taken as $i+1$ here and not i only because of 1-indexing for hostel numbers but 0-indexing in arrays.) All `sqend[i]`'s are initially null.

   Note that enqueueing requires pointers to all the sub-queue ends, because it may occur at a number of places in the queue, whereas dequeueing only requires a single pointer at the front.

   Enqueueing: When a new student `s` shows up to join the queue:

   - Obtain the hostel number `h` of `s`. This is a constant $\mathcal{O}(1)$ time operation if the hostel numbers are stored within the student nodes themselves.
   - If `sqend[h-1]` is not null, i.e., there is a sub-queue for hostel `h` in the current queue, then insert `s` in the mess queue, immediately after the student to which `sqend[h-1]` points.
   - If `sqend[h-1]` is null, then there is no sub-queue for the student's hostel, and he/she must be attached at the end of the entire queue. Using the `tail` pointer, insert `s` at the end. It may so happen that `tail` is null too. This means the entire queue is empty; in this case, simply make both `head` and `tail` point to `s`.
   - If `s` has been inserted at the end of the queue, then we have to change `tail` to point at `s`. This should be done if either `sqend[h-1]` is null (no sub-queue already present for `s`), or `sqend[h-1]` and `tail` point to the same student (sub-queue for `s` present at the end).
   - Update `sqend[h-1]` to point at `s`.

   All the above operations take constant $\mathcal{O}(1)$ time, so enqueueing is a constant time operation.

   Dequeueing: Dequeueing a student only happens at the front:

   - Use the `head` pointer to obtain the student `s` to be removed, and his/her hostel number `h`.

- Change `head` to point to the student immediately after `s` in the queue. If there are no more students in the queue, `head` becomes null.
- If the new `head` of the queue is either null, or has a hostel number different from `h`, that means the sub-queue of hostel `h` no longer exists after dequeueing `s`; set `sqend[h-1]` to null.
- If the new `head` of the queue is null, i.e., it's empty, set `tail` to null too.

All the above operations take constant $\mathcal{O}(1)$ time, so dequeueing is a constant time operation.

Now, think about the average waiting time in such a queue:

$$\text{average waiting time} = \frac{\text{sum of waiting times for all students in queue (total load)}}{\text{total number of students in queue}}$$

We make the (reasonable) assumption that dequeueing at the front of the queue takes place at a constant rate, say 1 person per unit time. Then, the waiting time for any student in the queue is equal to the number of students in front of him/her.

- In an ordinary queue of size n, a newcomer gets attached at the end. Then, the waiting time for the newcomer is n, and the waiting times of all the other students remain unchanged. The sum of waiting times (total load of the queue) increases by n.
- In a mess queue of size n which operates as in this problem, a newcomer may get attached at one of many positions in the queue. Suppose the newcomer is inserted after the first k students in the queue. Then, the waiting time for the newcomer is k, and there is an increase of 1 in the waiting times of all the $n - k$ students behind the newcomer. Thus, the increase in sum of waiting times in the queue is $k + (n - k) = n$.

Because the increase is the same in both cases, it can be easily seen that the average waiting time would also be the same; that is, the average waiting time in a queue is not influenced by the arrival process distribution. For more on this, see Little's Law[1].

[1] https://en.wikipedia.org/wiki/Little's_law