Problem 1

Consider the following program with two unspecified lines.

```
for j = 1 to n :
  (*)
  while i > 1 :
    print i
    (**)
  end while
end for
```

Give an asymptotic upper bound on the running time, in terms of n for the given program when the missing lines are specified as follows:

(a) $(*) : i = n \qquad (**) : i = i - 1$

(b) $(*) : i = n \qquad (**) : i = i/2$

(c) $(*) : i = j \qquad (**) : i = i - 2$

(d) $(*) : i = j \qquad (**) : i = i/2$

Solution

The for-loop will execute $O(n)$ times, the choice of $(*)$ and $(**)$ determine how many times the inner while-loop will execute. The innermost code takes $O(1)$ time to execute, as does every other line not associated with a loop. So in all cases, the running time will be $O(1) \times O(n) = O(n)$ times the number of executions of the inner while-loop.

(a) In this case the while-loop executes $O(n)$ times for each iteration of the for-loop, so the running time is bounded above by $O(n) \times O(n) = O(n^2)$.

(b) In this case the while-loop executes $O(\log n)$ times, so the running time is bounded above by $O(n) \times O(\log n) = O(n \log n)$.

(c) In this case the number of executions of the while-loop changes with each iteration of the for-loop: the while-loop executes $j/2 = O(j)$ times in each iteration. Since $j \leq n$ we could use $O(n)$ as an upper bound for the number of executions of the while-loop in each iteration of the for loop, giving us a running time of $O(n^2)$ as with (a). However, it may be possible to obtain a better upper bound by summing the for-loop executions individually. This would give us a total running time of $O(1) + O(2) + \ldots + O(n)$, but this is also $O(n^2)$.

(d) In this case the while-loop executes $O(\log j)$ times. Again, we could use the fact that $j \leq n$ to simplify, giving an upper bound of $O(\log n)$ iterations of the while loop, and an overall running time of $O(n \log n)$ as with (b). Can we do better by summing the for-loop executions individually? We observe that for $j \in [n/2, n]$, $\log j \in [\log n - 1, \log n]$, so at least $n/2$ executions of the for-loop will take $\Omega(\log n)$ time. Therefore $O(n \log n)$ is the best upper bound we can obtain.

Problem 2
Let $\Sigma = \{0, 1\}$

(a) Recursively define a function $\mathsf{str2num} : \Sigma^+ \to \mathbb{N}$ that converts a non-empty word over Σ to the number that one obtains by viewing the word as a binary number. For example $\mathsf{str2num}(1100) = 12$, $\mathsf{str2num}(0111) = 7$, $\mathsf{str2num}(0000) = 0$.

(b) Recursively define a function $\mathsf{num2str} : \mathbb{N} \to \Sigma^+$ that converts a number to its (shortest) binary representation. *Hint: you may want to use* div *and* %.

(c) Writing your functions as code in the natural way,

(i) Give an asymptotic upper bound in terms of $\mathrm{length}(()w)$ on the running time to compute $\mathsf{str2num}(w)$.

(ii) Give an asymptotic upper bound in terms of n on the running time to compute $\mathsf{num2str}(n)$.

Solution

(a) One approach:

- $\mathsf{str2num}(0) = 0$
- $\mathsf{str2num}(1) = 1$
- $\mathsf{str2num}(0w) = \mathsf{str2num}(w)$ for $w \in \Sigma^+$
- $\mathsf{str2num}(1w) = 2^{\mathrm{length}(w)} + \mathsf{str2num}(w)$ for $w \in \Sigma^+$

(b) One approach:

- $\mathsf{num2str}(0) = 0$
- $\mathsf{num2str}(1) = 1$
- $\mathsf{num2str}(n) = \mathsf{num2str}(n \text{ div } 2).\mathsf{num2str}(n \,\%\, 2)$ where . is string concatenation, for $n \geq 2$.

Solution (ctd)

(c) (i) The "natural" code is:

```
str2num(w) :
  if w = 0:
    return 0
  else if w = 1:
    return 1
  else if w = 0w':
    return str2num(w')
  else if w = 1w':
    return 2^{length(w')} + str2num(w')
```

The running time for each of these lines, excluding the recursive calls, is $O(1)$. Computing $\mathsf{length}(w)$ takes $O(\mathsf{length}(w))$ time unless we store w "smartly" using a complex data structure that keeps track of the length of w. If we let $T(n)$ denote the running time of $\mathsf{str2num}(w)$ when $\mathsf{length}(w) = n$ we see that the first recursive call (line 6) will take $O(1) + T(n-1)$ time; whereas the second call (line 8) will take $O(1) + O(n) + T(n-1)$ time. In the worst case, we will always execute the statement that takes the longest time giving us the following recurrence for $T(n)$:

$$T(1) \in O(1) \qquad T(n) \leq O(1) + O(n) + T(n-1).$$

Therefore, using the linear form of the Master Theorem, $T(n) \in O(n^2)$.

(ii) The "natural" code is:

```
num2str(n) :
  if n = 0:
    return 0
  else if n = 1:
    return 1
  else:
    return num2str(n div 2).num2str(n % 2)
```

Let $T(n)$ denote the running time of $\mathsf{num2str}(n)$. We observe that $\mathsf{num2str}(n \ \% \ 2)$ will execute in $O(1)$ time, and with a suitable method of storing words, concatenating the single symbol will also take $O(1)$ time. So the final line will take $T(n/2) + O(1)$ time to execute. For $n \geq 2$ this line will always get executed, giving us the following recurrence for $T(n)$:

$$T(0), T(1) \in O(1) \qquad T(n) \leq O(1) + T(n/2)$$

Using the Master Theorem, we have $a = 1$, $b = 2$, $c = d = 0$, so we are in Case 2, and $T(n) \in O(\log n)$.

Problem 3

Consider the procedure given in lectures to simulate a die using a fair coin:

(A) Flip a coin 3 times.

(B) If the outcome was:

- HHH: Output 1
- HHT: Output 2
- HTH: Output 3
- HTT: Output 4
- THH: Output 5
- THT: Output 6
- TTH: Go to (A)
- TTT: Go to (A)

What is the expected number of coin flips to obtain an output?

Solution

Let E be the expected number of coin flips (starting at (A)) before we output a number. With probability $\frac{6}{8} = \frac{3}{4}$ we will output something after 3 coin flips. With probability $\frac{2}{8} = \frac{1}{4}$ we will take 3 coin flips and return to (A) where we know we will take, on average, E more coin flips. So,

$$E = \frac{3}{4}.3 + \frac{1}{4}(3 + E).$$

Solving for E yields $E = 4$.

Problem 4

We want to tile a $2 \times n$ rectangle with 2×1 tiles so that the rectangle is completely covered and no tiles are overlapping. For example, here are two different ways to tile a 2×3 rectangle:

How many different ways (ignoring symmetry) are there of tiling a $2 \times n$ rectangle with 2×1 tiles in this way?

Solution

Let $T(n)$ be the number of ways of tiling a $2 \times n$ rectangle. We will find a formula for $T(n)$.
First we observe that $T(1) = 1$ and $T(2) = 2$.
For $n > 2$, let us consider how we fill the left-most positions in the tiling. We can either fill it with a single vertically oriented tile, or with 2 horizontally oriented tiles, and there are no other ways. Let us count how many ways there are of tiling in each of these cases. In the first case we have a $2 \times (n-1)$ rectangle remaining to tile, and we know how many ways there are of doing this: $T(n-1)$. In the second case we have a $2 \times (n-2)$ rectangle remaining to tile, and we can do this in $T(n-2)$ ways. So, in total, there are $T(n-1) + T(n-2)$ ways to tile a $2 \times n$ rectangle. That is $T(n) = T(n-1) + T(n-2)$. We can solve this: $T(n) = \textsc{Fib}_{n+1}$, the $(n+1)$-th Fibonacci number.

Problem 5

A tennis doubles match consists of two teams of two players per team. Ordering between teams, and within teams is not considered.

(a) How many different tennis doubles matches can be made with 4 players?

(b) How many different tennis doubles matches can be made with 5 players?

(c) How many different tennis doubles matches can be made from n players?

Solution

(a) Three possible approaches:

- Identify one player, A, say. We observe that the doubles match is completely determined once we choose A's partner, and there are 3 ways to do this.
- Take an arrangement of the four players. The first two in the arrangement make up one team and the second two make up the other team. There are $4! = 24$ arrangements of the four players, but we have duplication. Since the order in each team doesn't matter, we need to divide this by 2 for each team; and since the order of the teams does not matter we further divide by 2. So the total number of matches is $\frac{24}{2.2.2} = 3$.
- Let us first assume the teams are ordered/identified. To choose the players in the first team, we need to pick a subset of size 2. There are $\binom{4}{2} = 6$ ways of doing this. Once we have chosen the first team, the second team is determined, but we can also view it as choosing a subset of size 2 from the remaining 2 players: $\binom{2}{2} = 1$ possibility. This gives $6 \times 1 = 6$ ways of having ordered teams, so to account for the order of teams being irrelevant, we divide this by the number of ways of ordering the teams (2); giving a total of $\frac{6}{2} = 3$ matches.

(b) Three possible approaches:

- Choose one player to sit out. There are 5 ways of doing this. Once a player is sitting out, we know from the previous question that there are 3 ways to organise the match. Giving a total of $5 \times 3 = 15$ matches.
- Take an arrangement of the five players. The first two make up one team, the next two make up the second team, and the remaining person sits out. There are $5! = 120$ arrangements of the five players, but we need to account for the order within teams (divide by 2 for each team), and the order of the teams (divide by 2). This gives $\frac{120}{2.2.2} = 15$ matches.
- As with the previous question, there are $\binom{5}{2} = 10$ ways to choose the players in the first team; and $\binom{3}{2} = 3$ ways to choose the players in the second team. There are 2 ways of ordering the teams, so if the order does not matter there are $\frac{10\times 3}{2} = 15$ possible matches.

(c) Three possible approaches (note: all answers are the same, just presented differently):

- Choose $(n-4)$ players to sit out (equivalently 4 players to play). This can be done in $\binom{n}{n-4} = \binom{n}{4}$ ways. Once the players have been chosen, there are 3 ways of arranging them as shown in (a). Giving the total number of matches as $3 \times \binom{n}{4}$.
- Take an arrangement of the n players. The first two make up one team, the next two make up the second team, and the remaining $n-4$ players sit out. There are $n!$ arrangements of the n players, but we need to account for the order within teams (divide by 2 for each team); the order of the teams (divide by 2); and the order of the people sitting out (divide by $(n-4)!$). This gives the total number of matches as $\frac{n!}{2.2.2.(n-4)!} = \frac{\Pi(n,4)}{8}$.
- Assume an ordering on the teams. There are $\binom{n}{2}$ ways to choose the first team, and $\binom{n-2}{2}$ ways to choose the second team. Dividing by 2 to account for the ordering of the teams gives the total number of matches as $\frac{1}{2}\binom{n}{2}\binom{n-2}{2}$.