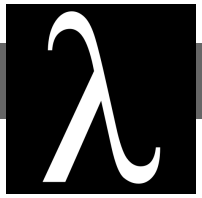

Contents

1 Work/Span Lab

1.1 Work and Span

- **Work:** The number of steps to sequentially evaluate code.
- **Span:** Assuming infinite processors, the number of steps to evaluate code in parallel.

We can express and solve for the work and span of recursive functions with *recurrences*. One way to gain intuition about the summation form of a recurrence is the *tree method* [1].

1.1.1 Tree Method for Finding Work/Span

Tree Method:

1. Write the recurrence - base case and recursive case, if applicable.
2. Identify the number of levels (L), nodes at level i (n_i), and the non-recursive work/span per node at level i (w_i).
3. Solve the summation $\sum_{i=0}^{L} n_i * w_i$.
4. Use this summation to find the big-O bound.

[1] While it may seem easy to demonstrate recurrence bounds by induction, these proofs are trickier than they seem! The nature of a bound makes such proofs hard to write correctly, so we strongly recommend you use the tree method on assignments and exams.

1.1.2 Writing and Solving Recurrences

Task 1.1.

```
fun one 0 = 1
  | one n = one (n div 2)
```

$W_{\texttt{one}}(0) =$

$W_{\texttt{one}}(n) =$

$W_{\texttt{one}}(n) \in O(\quad)$

```
fun min [] = 0
  | min (x::xs) =
      Int.min (x, min xs)
```

$W_{\texttt{min}}(0) =$

$W_{\texttt{min}}(n) =$

$W_{\texttt{min}}(n) \in O(\quad)$

1.1.3 Common Recurrences

Below is a table of asymptotic bounds on common recurrences.

Recurrence			Asymptotic Upper Bound
$T(n)$	$=$	$T(n-1)+c$	$O(n)$
$T(n)$	$=$	$T(n/2)+c$	$O(\log n)$
$T(n)$	$=$	$2T(n/2)+c$	$O(n)$
$T(n)$	$=$	$T(n/2)+c_1 n+c_0$	$O(n)$
$T(n)$	$=$	$2T(n/2)+c_1 n+c_0$	$O(n \log n)$
$T(n)$	$=$	$T(n-1)+c_1 n+c_0$	$O(n^2)$
$T(n)$	$=$	$2T(n-1)+c$	$O(2^n)$

Note: You may not simply cite the table of recurrences given in lab to justify the work and span bounds you derive on the homework or exams. While the table may be helpful for double checking that your work is indeed correct, you must ***always*** show the steps you take to get from your recurrence to your final bound.

1.1.4 Common Summations

Here are some formulas that you may find useful when trying to solve summations:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{1}$$

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

$$\sum_{i=0}^{\log n} \frac{1}{2^i} \leq \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \in O(1) \tag{3}$$

$$\sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a} \qquad \text{(generalization of 1)} \tag{4}$$

1.2 Work and Span for Trees

1.2.1 Balanced vs. Unbalanced

You may only assume the tree is balanced if it is stated in the problem statement. Otherwise, you must analyze the cost bounds given the worst-case scenario.

1.2.2 Nodes vs. Depth

When the input to a function is a tree, we can analyze the work and span through either the number of nodes in the tree, n, or the depth of the tree, d.

Task 1.2.

For the following questions, consider both input sizes (number of nodes, n, and depth, d).

Given a "standard" tree function (i.e. one that makes two recursive calls on the left and right subtree), what would be the work recurrences for a balanced tree?

What about a maximally unbalanced tree (e.g. each node only has children in its left subtree)?

Task 1.3.

```
datatype tree = Empty | Node of tree * int * tree

fun insert (Empty : tree, n : int) : tree = Node (Empty, n, Empty)
  | insert (Node (L, x, R) : tree, n : int) : tree =
      if n <= x then insert (L, x)
      else insert (R, x)
```

What is the work and span of `insert` in terms of the number of nodes in the tree, n? Assume that the tree is balanced.

What are the work and span of `insert` in terms of the depth of our tree, d? Do not assume that the tree is balanced.

2 Mult

2.1 Analyzing `mult`

Recall the tree datatype:

```
datatype tree = Empty | Node of tree * int * tree
```

Consider this function, which computes the product of integers in a tree:

```
fun mult Empty = 1
  | mult (Node(L,x,R)) = x * mult L * mult R
```

Note that function application takes precedence over infix operators such as `*`.

For this section, you may assume that the trees are *balanced*, meaning that the tree is either an empty tree or a tree whose subtrees are themselves balanced and whose subtrees differ in height by at most one.

Task 2.1.

Given that `T` is a balanced `tree` with depth d, write recurrences for the work and span of evaluating `mult T`, in terms of d.

Task 2.2.

Given that `T` is a balanced `tree` with n nodes, write recurrences for the work and span of evaluating `mult T`, in terms of n.

Task 2.3.

Using the recurrences you found in the previous tasks, what are the big-O bounds for the work and the span respectively? Show your work. You should have 2 big-O bounds in terms of d (one for work and one for span) and 2 big-O bounds in terms of n (one for work and one for span).

3 Trees

In the Datatypes Lab, you wrote the function `invert`, which inverts a tree so that the in-order traversal of the inversion comes out backwards:

```
fun invert (Empty : tree) : tree = Empty
  | invert (Node (L, x, R)) = Node (invert R, x, invert L)
```

In this problem, you will analyze the work and span of this function.

Task 3.1.

Determine the recurrence for the work of `invert`, in terms of the size (number of elements) of the tree. You may assume the tree is balanced.

Task 3.2.

Use the closed form of the recurrence from the previous step to determine the big-O of $W_{\texttt{invert}}(n)$.

Task 3.3.

Determine the recurrence for the span of your `invert` function, in terms of the size of the tree. You may assume the tree is balanced.

Task 3.4.

Use the closed form of the recurrence from the previous step to give a big-O for $S_{\texttt{invert}}$.

4 Work It Out

Consider the following two functions:

```
fun listMax ([] : int list) : int = 0
  | listMax (x::xs) = Int.max (x, listMax xs)

fun treeMax (Empty : tree) : int = 0
  | treeMax (Node (l,x,r)) =
      Int.max (treeMax l, Int.max (x, treeMax r))
```

For each of the functions below:

- Write the function's recurrence relations.
- Solve the recurrence with tree method.
- Derive a tight big-O bound.
- Briefly explain why your answer is correct.

You may not use the table of common recurrences for these problems.

Task 4.1.

Find the *work* of `listMax`, in terms of the length, n.

Task 4.2.

Find the *span* of `listMax`, in terms of the length, n.

For the following tasks (4.3 - 4.6), assume the tree is balanced.

Task 4.3. (Recommended)

Find the *work* of `treeMax`, in terms of the number of nodes in the tree, n.

Task 4.4. (Recommended)

Find the *span* of `treeMax`, in terms of the number of nodes in the tree, n.

Task 4.5.

Find the *work* of `treeMax`, in terms of the tree's depth, d.

Task 4.6.

Find the *span* of `treeMax`, in terms of the tree's depth, d.

For the following tasks (4.7 - 4.11), assume the tree is maximally unbalanced.

Task 4.7. (Recommended)

Find the *work* of `treeMax`, in terms of the number of nodes in the tree, n.

Task 4.8.

You should have gotten the same asymptotic bound for the work of `treeMax` in both the balanced case and the maximally unbalanced case. It turns out that the work of `treeMax` only depends on n, the number of nodes in the tree, and not the structure of the tree. Why is this the case? (A brief justification suffices.)

Hint: Suppose there are n_l nodes in the left subtree and n_r nodes in the right subtree. Write the recurrence for this setup.

Task 4.9. (Recommended)

Find the *span* of `treeMax`, in terms of the number of nodes in the tree, n.

Task 4.10.

Find the *work* of `treeMax`, in terms of the tree's depth, d.

Task 4.11.

Find the *span* of `treeMax`, in terms of the tree's depth, d.

Task 4.12.

Compare your answers to 4.5 and 4.10. Which kind of tree (balanced or maximally unbalanced) is the worst case for `treeMax` when analyzed in terms of depth?

How can we determine the worst case for `treeMax` without doing both analyses?