Contents

1 Work and Span 2
1.1 Work and Span ............................................. 2
1.2 Writing and Solving Recurrences .............................. 2
    1.2.1 Common Recurrences .................................. 3

2 Work It Out 4
Task 1. ................................................................. 4
Task 2. ................................................................. 4
Task 3. ................................................................. 4
Task 4. ................................................................. 4
Task 5. ................................................................. 4
Task 6. ................................................................. 4
Task 7. ................................................................. 5
Task 8. ................................................................. 5
Task 9. ................................................................. 5
Task 10. ............................................................... 5

3 Lists: The Electric Boogaloo 6
Task 11. .............................................................. 6
Task 12. .............................................................. 6
Task 13. .............................................................. 6

4 Trees 7
Task 14. .............................................................. 7
Task 15. .............................................................. 7
Task 16. .............................................................. 7
Task 17. .............................................................. 7
Task 18. .............................................................. 7
Task 19. .............................................................. 7
Task 20. .............................................................. 7
1 Work and Span

1.1 Work and Span

- **Work:** The number of steps it takes 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. Two ways to gain intuition about the bound of a recurrence are the tree method and unrolling.

1. Write the recurrence.
2. Examine the recurrence (draw a tree, unroll, write a summation).
3. Determine a tight big-$O$ bound.

1.2 Writing and Solving Recurrences

```plaintext
fun treeSum Empty = 0
| treeSum (Node (L, x, R)) = treeSum L + x + treeSum R
```

First, let’s analyze `treeSum` in terms of $n$, the number of nodes in the tree. You may assume the tree is balanced.

\[
\begin{align*}
W_{\text{treeSum}}(0) &= \\
W_{\text{treeSum}}(n) &= \\
W_{\text{treeSum}}(n) &\in O(\ )
\end{align*}
\]

\[
\begin{align*}
S_{\text{treeSum}}(0) &= \\
S_{\text{treeSum}}(n) &= \\
S_{\text{treeSum}}(n) &\in O(\ )
\end{align*}
\]

Now let’s analyze `treeSum` in terms of $d$, the depth of the tree. You may assume the tree is balanced. Remember than $2^d = n$.

\[
\begin{align*}
W_{\text{treeSum}}(0) &= \\
W_{\text{treeSum}}(d) &= \\
W_{\text{treeSum}}(d) &\in O(\ )
\end{align*}
\]

\[
\begin{align*}
S_{\text{treeSum}}(0) &= \\
S_{\text{treeSum}}(d) &= \\
S_{\text{treeSum}}(d) &\in O(\ )
\end{align*}
\]
fun inord Empty = []
  | inord (Node (l, x, r)) = inord l @ (x::inord r)

Let’s analyze \texttt{inord} in terms of \(n\), the number of nodes in the tree. You may assume the tree is balanced.

\[
W_{\text{inord}}(0) = \\
W_{\text{inord}}(n) = \\
W_{\text{inord}}(n) \in O( )
\]

\[
S_{\text{inord}}(0) = \\
S_{\text{inord}}(n) = \\
S_{\text{inord}}(n) \in O( )
\]

Now let’s analyze \texttt{inord} in terms of \(d\), the depth of the tree. You may assume the tree is balanced.

\[
W_{\text{inord}}(0) = \\
W_{\text{inord}}(d) = \\
W_{\text{inord}}(d) \in O( )
\]

\[
S_{\text{inord}}(0) = \\
S_{\text{inord}}(d) = \\
S_{\text{inord}}(d) \in O( )
\]

### 1.2.1 Common Recurrences

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Asymptotic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T(n) = T(n-1)+c)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(T(n) = T(n/2)+c)</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>(T(n) = 2T(n/2)+c)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(T(n) = T(n/2)+c_1n+c_0)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(T(n) = 2T(n/2)+c_1n+c_0)</td>
<td>(O(n\log n))</td>
</tr>
<tr>
<td>(T(n) = T(n-1)+c_1n+c_0)</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>(T(n) = 2T(n-1)+c)</td>
<td>(O(2^n))</td>
</tr>
</tbody>
</table>
2 Work It Out

Consider the following two functions:

```plaintext
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.
• Convert the recurrence into a summation.
• 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. You may use the fact that:

\[
\sum_{i=0}^{n-1} a^i = \frac{1 - a^n}{1 - a}
\]

Task 1.
Find the \textit{work} of listMax, in terms of the length, \(n\).

Task 2.
Find the \textit{span} of listMax, in terms of the length, \(n\).

Task 3.
Find the \textit{work} of treeMax, in terms of the number of nodes in the tree, \(n\).

Task 4.
Find the \textit{span} of treeMax, in terms of the number of nodes in the tree, \(n\).

Task 5.
Find the \textit{work} of treeMax, in terms of the tree’s depth, \(d\).

Task 6.
Find the \textit{span} of treeMax, in terms of the tree’s depth, \(d\).
For the following tasks, do not assume the tree is balanced.

Task 7.
Find the work of $\text{treeMax}$, in terms of the number of nodes in the tree, $n$.

Task 8.
Find the span of $\text{treeMax}$, in terms of the number of nodes in the tree, $n$.

Task 9.
Find the work of $\text{treeMax}$, in terms of the tree’s depth, $d$.

Task 10.
Find the span of $\text{treeMax}$, in terms of the tree’s depth, $d$.  

3 Lists: The Electric Boogaloo

Task 11.

In code/lists/lists.sml, define

```
reverse : int list -> int list
REQUIRES: true
ENSURES: reverse L ⇒ a list that contains all the elements of L in reverse order
```

For this task you should define a helper function that will allow you to solve the problem in $O(n)$ work.

Task 12.

In code/lists/lists.sml, write a function

```
merge : int list * int list -> int list
REQUIRES: l1 and l2 are sorted
ENSURES: merge (l1, l2) ⇒ l where l is a sorted permutation of l1 @ l2
```

where sorted means non-descending order.

Task 13.

Write a recurrence relation for the work of merge, in terms of the lengths of l1 and l2. What is the big-$O$ of this recurrence?
4 Trees

Here is a function similar to the flatten function from lecture, which computes an in-order traversal of a tree:

```sml
fun treeToList (Empty : tree) : int list = []
    | treeToList (Node (l,x,r)) = treeToList l @ (x :: (treeToList r))
```

Observe that `treeToList` is total.

In this problem, you will define a function to mirror, or reverse a tree, so that the in-order traversal of the reverse comes out backwards:

\[ \text{treeToList} \left( \text{revT} \ t \right) \equiv \text{reverse} \left( \text{treeToList} \ t \right) \]

**Task 14.**

In `code/reverse/reverse.sml`, define the function

```sml
revT : tree -> tree
REQUIRES: true
ENSURES: treeToList (revT t) = reverse (treeToList t)
```

For example, `val Node(Node(Empty,1,Empty),5,Node(Empty,0,Empty)) = revT (Node(Node(Empty,0,Empty),5,Node(Empty,1,Empty)))`

**Task 15.**

Prove that `revT` is total.

**Task 16.**

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

**Task 17.**

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

**Task 18.**

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

**Task 19.**

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

**Task 20.**

Prove the following:
Theorem. For all values \( t : \text{tree} \),

\[
\text{treeToList (revT t)} = \text{reverse (treeToList t)}.
\]

You may use the following lemmas about \text{reverse} on lists:

- \( \text{reverse \ [\]} = \[\] \)
- For all expressions \( l \) and \( r \) of type \text{int list} such that \( l \) and \( r \) both reduce to values,

\[
\text{reverse \ (l @ (x::r))} = \text{reverse \ r @ (x :: reverse \ l)}
\]

In your justifications, be careful to prove that expressions evaluate to values when this is necessary. Follow the template on the following page.

Case for Empty
To show:

Case for Node \((l,x,r)\)
Inductive hypothesis 1:
Inductive hypothesis 2:

To show: