Lecture 23. Recursive Data Structure: Algorithm Complexity

Efficiency by Complexity of Algorithms

- Apparently, the BSearch function takes less work (and time) to evaluate than the ASearch function. How much better is it?

Efficiency by Complexity of Algorithms

- Apparently, the BSearch function takes less work (and time) to evaluate than the ASearch function. How much better is it?
- Let's count the number of occurences of the most time-consuming operation, the comparisons $t = x$ and $t > r$.
  - **ASearch function** $\Theta(n)$ for n elements in a list.
  - **BSearch function** $\Theta(\log_2 n)$ for n elements in a list.
  - As the size of the list gets large, BSearch becomes a much better alternative to ASearch. For example, a list containing 1,048,576 elements requires 1,048,576 comparisons using ASearch, but only 21 comparisons using BSearch.

Explanation: Complexity of "ASearch" Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.

Explanation: Complexity of "ASearch" Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.
- With a Slist of the depth p, $n = 2^p$. So, we may set $C(n) = \tilde{C}(p)$

Explanation: Complexity of "ASearch" Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.
- With a Slist of the depth p, $n = 2^p$. So, we may set $C(n) = \tilde{C}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{C}(0) = 1$.

Explanation: Complexity of “ASearch” Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.
- With a Slist of the depth p, $n = 2^p$. So, we may set $C(n) = \tilde{C}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{C}(0) = 1$.
- Now suppose the ASearch function is evaluated on a list of depth p for some $p > 0$. Then the recursive case **R** of the definition gets used, and the ASearch function is executed twice on a list of depth $p - 1$.

Explanation: Complexity of “ASearch” Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.
- With a Slist of the depth p, $n = 2^p$. So, we may set $C(n) = \tilde{C}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{C}(0) = 1$.
- Now suppose the ASearch function is evaluated on a list of depth p for some $p > 0$. Then the recursive case **R** of the definition gets used, and the ASearch function is executed twice on a list of depth $p - 1$.
- Each of these two recursive calls uses $\tilde{C}(p - 1)$ comparisons.

Explanation: Complexity of "ASearch" Algorithm

- Let $C(n)$ be the number of times the comparison $t = x$ is done when searching a list of n elements.
- With a Slist of the depth p, $n = 2^p$. So, we may set $C(n) = \tilde{C}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{C}(0) = 1$.
- Now suppose the ASearch function is evaluated on a list of depth p for some $p > 0$. Then the recursive case **R** of the definition gets used, and the ASearch function is executed twice on a list of depth $p - 1$.
- Each of these two recursive calls uses $\tilde{C}(p-1)$ comparisons.
- Hence $\tilde{C}(p)$ must satisfy the following recurrence relation

$$\tilde{C}(p) = \begin{cases} 1 & \text{if} \quad p = 0 \\ 2\tilde{C}(p-1) & \text{if} \quad p > 0 \end{cases}$$

- The closed-form solution of the recurrence relation is $\tilde{C}(p) = 2^p$.

- The closed-form solution of the recurrence relation is $\tilde{C}(p) = 2^p$.
- So, a list of 2^p requires approximately 2^p comparisons, in the worst case.

- The closed-form solution of the recurrence relation is $\tilde{C}(p) = 2^p$.
- So, a list of 2^p requires approximately 2^p comparisons, in the worst case.
- In general, if a list has n elements, then $C(n) = n$ since $n = 2^p$. So we can approximate the worst-case complexity of this algorithm as $\Theta(n)$.

Explanation: Complexity of "BSearch" Algorithm

- Let $D(n)$ be the number of times the comparison $t > r$ is done when searching a list of n elements.
- In this case of Slists, $n = 2^p$. So, let $D(n) = \tilde{D}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{D}(0) = 1$.

Explanation: Complexity of "BSearch" Algorithm

- Let $D(n)$ be the number of times the comparison $t > r$ is done when searching a list of n elements.
- In this case of Slists, $n = 2^p$. So, let $D(n) = \tilde{D}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{D}(0) = 1$.
- Now suppose the BSearch function is evaluated on a list of depth p for some $p > 0$. Then the recursive part **R** first makes one comparison, and then calls the BSearch function on a list of depth $p - 1$, which uses a $\tilde{D}(p - 1)$ comparisons.

Explanation: Complexity of "BSearch" Algorithm

- Let $D(n)$ be the number of times the comparison $t > r$ is done when searching a list of n elements.
- In this case of Slists, $n = 2^p$. So, let $D(n) = \tilde{D}(p)$
- If $p = 0$, then the list contains only a single item so that the base case **B** of the definition gets used and one comparison is made. Therefore, $\tilde{D}(0) = 1$.
- Now suppose the BSearch function is evaluated on a list of depth p for some $p > 0$. Then the recursive part **R** first makes one comparison, and then calls the BSearch function on a list of depth $p - 1$, which uses a $\tilde{D}(p-1)$ comparisons.
- Hence $\tilde{D}(p)$ must satisfy the following recurrence relation

$$\tilde{D}(p) = \begin{cases} 1 & \text{if} \quad p = 0 \\ 1 + \tilde{D}(p-1) & \text{if} \quad p > 0 \end{cases}$$

- The closed-form solution of the recurrence relation is $\tilde{D}(p) = p + 1$.

- The closed-form solution of the recurrence relation is $\tilde{D}(p) = p + 1$.
- So, we only need $p + 1$ comparisons to find an element in a list of size 2^p.
- So, a list of 2^p requires approximately $p + 1$ comparisons, in the worst case.
- Substituting $p = \log_2 n$ for some p, we get $D(n) = \log_2 n + 1$, so we can approximate the worst-case complexity of this algorithm as $\Theta(\log_2 n)$.

Activity 23.1. Let L be an SList. Define a recursive function Wham as follows.

B. Suppose $L = x$. Then Wham(L) $= x \cdot x$.

R. Suppose $L = (X, Y)$. Then, Wham(L) $=$ Wham(X) $+$ Wham(Y).

(a) Evaluate Wham(((1,2), (4,5))), showing all steps.

(b) Give a recurrence relation for $S(p)$, the number of $+$ operations performed by Whom on an SList of depth p, for $p > 0$.

(c) Give a recurrence relation for $M(p)$, the number of $\cdot$ operations performed by Whom on an SList of depth p, for $p > 0$.

Exercise. The following is an recursive binary search algorithm

---

Algorithm 5.4 Binary Search (recursive).

Preconditions: The set U is totally ordered by $<$, and $X = \{x_1, x_2, \ldots, x_n\} \subseteq U$, with $n \geq 1$,

$$x_1 < x_2 < \cdots < x_n,$$

and $t \in U$. Also, $1 \leq l \leq r \leq n$.

Postconditions: $\texttt{BinSearch}(t, X, l, r) = (t \in \{x_l, x_{l+1}, \ldots, x_r\})$

```
function BinSearch(t ∈ U,
                   X = {x_1, x_2, ..., x_n} ⊆ U,
                   l, r ∈ {1, 2, ..., n})
   i ← ⌊(l + r)/2⌋
   if t = x_i then
      return true
   else
      if (t < x_i) ∧ (l < i) then
         return BinSearch(t, X, l, i − 1)
      else
         if (t > x_i) ∧ (i < r) then
            return BinSearch(t, X, i + 1, r)
         else
            return false
```

(Continue to the next page)

A typical call to this function would look like this:

```
if BinSearch(t,{3,6,9,12,15},1,n) then
    print Element t was found.
else
    print Element t was not found.
```

The choice of 1 and n for the last two parameters tell the function to search the whole array. The following top-down evaluation of the recursive binary search looks for the target value 21 in the array $X = \{3, 6, 9, 12, 15, 18, 21, 24, 27, 30\}$.

$$
\begin{aligned}
\texttt{BinSearch}(21,X,1,10) &= \texttt{BinSearch}(21,X,6,10) \\
&= \texttt{BinSearch}(21,X,6,7) \\
&= \texttt{BinSearch}(21,X,7,7) \\
&= \texttt{true}
\end{aligned}
$$

(a) Evaluate BinSearch(3,X,1,10) using a top-dowm evaluation. (Write all the steps of evaluation.)

(b) Approximate the best- and worst-case complexity of the recursive binary search function.