The Selection Problem

The selection problem:

- given an integer \( k \) and a list \( x_1, \ldots, x_n \) of \( n \) elements
- find the \( k \)-th smallest element in the list

Example: the 3rd smallest element of the following list is 6

```
7 4 9 6 2
```

An \( O(n \cdot \log n) \) solution:

- sort the list \( (O(n \cdot \log n)) \)
- pick the \( k \)-th element of the sorted list \( (O(1)) \)

```
2 4 6 7 9
```

Can we find the \( k \)-th smallest element faster?
Quick-Select

Quick-select of the \( k \)-th smallest element in the list \( S \):

- based on **prune-and-search** paradigm
- Prune: pick random element \( x \) (pivot) from \( S \) and split \( S \) in:
  - \( L \) elements \(< x \), \( E \) elements \(= x \), \( G \) elements \(> x \)

\[
\begin{array}{cccccccc}
7 & 2 & 1 & 9 & 6 & 5 & 3 & 8 \\
2 & 1 & 3 & 5 & 7 & 9 & 6 & 8 \\
\end{array}
\]

- partitioning into \( L \), \( E \) and \( G \) works precisely as for quick-sort

- Search:
  - if \( k \leq |L| \) then return \( \text{quickSelect}(k, L) \)
  - if \( |L| < k \leq |L| + |E| \) then return \( x \)
  - if \( k > |L| + |E| \) then return \( \text{quickSelect}(k - |L| - |E|, G) \)
Quick-Select Visualization

Quick-select can be displayed by a sequence of nodes:

- each node represents recursive call and stores: $k$, the sequence, and the pivot element

$k = 5, \ S = (7 \ 4 \ 9 \ 3 \ 2 \ 6 \ 5 \ 1 \ 8)$

$k = 2, \ S = (7 \ 4 \ 9 \ 6 \ 5 \ 8)$

$k = 2, \ S = (7 \ 4 \ 6 \ 5)$

$k = 1, \ S = (7 \ 6 \ 5)$

found 5
Quick-Select: Running Time

The worst-case running is $O(n^2)$ time:
- if the pivot is always the minimal or maximal element

The expected running time is $O(n)$ (compare with quick-sort):
- with probability 0.5 the recursive call is good: $(3/4)n$ size
- $T(n) \leq b \cdot a \cdot n + T\left(\frac{3}{4}n\right)$
  - $a$ is the time steps for partitioning per element
  - $b$ is the expected number of calls until a good call
  - $b = 2$ (average number of coins to toss until head shows up)
- Let $m = 2 \cdot a \cdot n$, then
  
  $T(n) \leq 2 \cdot a \cdot n + T\left(\frac{3}{4}n\right)$
  
  $\leq 2 \cdot a \cdot n + 2 \cdot a \cdot (3/4) \cdot n + 2 \cdot a \cdot (3/4)^2 \cdot n \ldots$
  
  $= 8 \cdot a \cdot n \in O(n)$ (geometric series)
Quick-Select: Median of Medians

We can do selection in $O(n)$ worst-case time.

Idea: recursively use select itself to find a good pivot:

- divide $S$ into $n/5$ sets of 5 elements
- find a median in each set (baby median)
- recursively use select to find the median of the medians

The minimal size of $L$ and $G$ is $0.3 \cdot n$. 
Quick-Select: Median of Medians

We know:
- The minimal size of $L$ and $G$ is $0.3 \cdot n$.
- Thus the maximal size of $L$ and $G$ is $0.7 \cdot n$.

Let $b \in \mathbb{N}$ such that:
- partitioning of a list of size $n$ takes at most $b \cdot n$ time,
- finding the baby medians takes at most $b \cdot n$ time,
- the base case $n \leq 1$ takes at most $b$ time.

We derive a recurrence equation for the time complexity:

$$T(n) = \begin{cases} 
    b & \text{if } n \leq 1 \\
    T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n & \text{if } n > 1 
\end{cases}$$

We will see how to solve recurrence equations...
We have considered algorithms for solving special problems. Now we consider a few fundamental techniques:

- Devide-and-Conquer
- The Greedy Method
- Dynamic Programming
**Divide-and-Conquer** is a general algorithm design paradigm:

- **Divide:** divide input $S$ into $k \geq 2$ disjoint subsets $S_1, \ldots, S_k$
- **Recur:** recursively solve the subproblems $S_1, \ldots, S_k$
- **Conquer:** combine solutions for $S_1, \ldots, S_k$ to solution for $S$

(If the base case of the recursion are problems of size 0 or 1).

We have already seen examples:

- merge sort
- quick sort
- bottom-up heap construction

Focus now: analysing time complexity by recurrence equations.
We consider 3 methods for solving recurrence equation:

- Iterative substitution method,
- Recursion tree method,
- Guess-and-test method, and
- Master method.
Iterative Substitution

Iterative substitution technique works as follows:

- iteratively apply the recurrence equations to itself
- hope to find a pattern

Example

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n-1) + 2 & \text{if } n > 1
\end{cases} \]

We start with \( T(n) \) and apply the recursive case:

\[
T(n) = T(n-1) + 2 \\
= T(n-2) + 4 \\
= T(n-k) + 2 \cdot k
\]

For \( k = n - 1 \) we reach the base case \( T(n-k) = 1 \) thus:

\[ T(n) = 1 + 2 \cdot (n-1) = 2 \cdot n - 1 \]
Merge-Sort Review

Merge-sort of a list $S$ with $n$ elements works as follows:

- **Divide:** divide $S$ into two lists $S_1$, $S_2$ of $\approx n/2$ elements
- **Recur:** recursively sort $S_1$, $S_2$
- **Conquer:** merge $S_1$ and $S_2$ into a sorting of $S$

Let $b \in \mathbb{N}$ such that:

- merging two lists of size $n/2$ takes at most $b \cdot n$ time, and
- the base case $n \leq 1$ takes at most $b$ time

We obtain the following recurrence equation for merge-sort:

$$T(n) = \begin{cases} 
  b & \text{if } n \leq 1 \\
  2 \cdot T(n/2) + b \cdot n & \text{if } n > 1 
\end{cases}$$

We search a **closed solution** for the equation, that is:

- $T(n) = \ldots$ where $T(n)$ does not occur in the right side
Example: Merge-Sort

\[ T(n) = \begin{cases} 
  b & \text{if } n \leq 1 \\
  2 \cdot T(n/2) + b \cdot n & \text{if } n > 1 
\end{cases} \]

We assume that \( n \) is a power of 2: \( n = 2^k \) (that is, \( k = \log_2 n \))

► allowed since we are interested in asymptotic behaviour

We start with \( T(n) \) and apply the recursive case:

\[ T(n) = 2 \cdot T(n/2) + b \cdot n \]

\[ = 2 \cdot (2 \cdot T(n/2^2) + b \cdot (n/2)) + b \cdot n \]

\[ = 2^2 \cdot T(n/2^2) + 2 \cdot b \cdot n \]

\[ = 2^3 \cdot T(n/2^3) + 3 \cdot b \cdot n \]

\[ = 2^k \cdot T(n/2^k) + k \cdot b \cdot n \]

\[ = n \cdot b + (\log_2 n) \cdot b \cdot n \]

Thus \( T(n) = b \cdot (n + n \cdot \log_2 n) \in O(n \log_2 n) \).
The Recursion Tree Method

The recursion tree method is a visual approach:

- draw the recursion tree and hope to find a pattern

\[
T(n) = \begin{cases} 
  b & \text{if } n \leq 2 \\
  3 \cdot T(n/3) + b \cdot n & \text{if } n > 2 
\end{cases}
\]

For a node with input size \( k \): work at this node is \( b \cdot k \).

<table>
<thead>
<tr>
<th>depth</th>
<th>nodes</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>( n/3 )</td>
</tr>
<tr>
<td>( i )</td>
<td>( 3^i )</td>
<td>( n/3^i )</td>
</tr>
</tbody>
</table>

Thus the work at depth \( i \) is \( 3^i \cdot b \cdot n/3^i = b \cdot n \).

The height of the tree is \( \log_3 n \). Thus \( T(n) \) is \( O(n \cdot \log_3 n) \).
Guess-and-Test Method

The guess-and-test method works as follows:

- we guess a solution (or an upper bound)
- we prove that the solution is true by induction

Example

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n/2) + 2 \cdot n & \text{if } n > 1 
\end{cases} \]

Guess: \( T(n) \leq 2 \cdot n \)

- for \( n = 1 \) it holds \( T(n) = 1 \leq 2 \cdot 1 \)
- for \( n > 1 \) we have:
  \[ T(n) = T(n/2) + 2 \cdot n \leq 2 \cdot n/2 + 2 \cdot n = 3 \cdot n \]

Wrong guess: we cannot make \( 3 \cdot n \) smaller or equal to \( 2 \cdot n \).
Example, continued

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n/2) + 2 \cdot n & \text{if } n > 1 
\end{cases}
\]

New guess: \( T(n) \leq 4 \cdot n \)

- for \( n = 1 \) it holds \( T(n) = 1 \leq 4 \cdot 1 \)

- for \( n > 1 \) we have:

\[
T(n) = T(n/2) + 2 \cdot n \\
\leq 4 \cdot n/2 + 2 \cdot n \quad \text{(by induction hypothesis)} \\
= 4 \cdot n
\]

This time the guess was good: \( 4 \cdot n \leq 4 \cdot n \). Thus \( T(n) \leq 4 \cdot n \).
Example: Quick-Select with Median of Median

\[ T(n) = \begin{cases} 
  b & \text{if } n \leq 1 \\
  T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n & \text{if } n > 1 
\end{cases} \]

Guess: \( T(n) \leq 20 \cdot b \cdot n \)

- for \( n = 1 \) it holds \( T(n) = b \leq 20 \cdot b \cdot 1 \)
- for \( n > 1 \) we have:

\[
T(n) = T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n \\
\leq 0.7 \cdot 20 \cdot b \cdot n + 0.2 \cdot 20 \cdot b \cdot n + 2 \cdot b \cdot n \quad \text{(by IH)} \\
= 0.9 \cdot 20 \cdot b \cdot n + 2 \cdot b \cdot n = 18 \cdot b \cdot n + 2 \cdot b \cdot n \\
= 20 \cdot b \cdot n
\]

Thus the guess was good.
This shows that quick-select with median of median is \( O(n) \).
Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} 
    c & \text{if } n < d \\
    aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}$$

**Theorem (The Master Theorem)**

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a \log^k n})$, then $T(n)$ is $\Theta(n^{\log_b a \log^{k+1} n})$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$. 


Master Method: Example 1

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

Theorem (The Master Theorem)

1. If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[ T(n) = 4 \cdot T(n/2) + n \]

Solution: \( \log_b a = 2 \), thus case 1 says \( T(n) = \Theta n^2 \).
Master Method: Example 2

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

Theorem (The Master Theorem)

1. if \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[ T(n) = 2 \cdot T(n/2) + n \cdot \log n \]

Solution: \( \log_b a = 1 \), thus case 2 says \( T(n) = \Theta(n \log^2 n) \).
Master Method: Example 3

\[
T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d
\end{cases}
\]

Theorem (The Master Theorem)

1. if \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[
T(n) = T(n/3) + n \cdot \log n
\]

Solution: \( \log_b a = 0 \), thus case 3 says \( T(n) = \Theta(n \log n) \).
Master Method: Example 4

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. If \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided 
\[ af(n/b) \leq \delta f(n) \] for some \( \delta < 1 \).

**Example**

\[ T(n) = 8 \cdot T(n/2) + n^2 \]

Solution: \( \log_b a = 3 \), thus case 1 says \( T(n) = \Theta(n^3) \).
Master Method: Example 5

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

**Example**

\[ T(n) = 9 \cdot T(n/3) + n^3 \]

Solution: \( \log_b a = 2 \), thus case 3 says \( T(n) = \Theta(n^3) \).
Master Method: Example 6

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

**Example**

\[ T(n) = T(n/2) + 1 \quad \text{binary search} \]

Solution: \( \log_b a = 0 \), thus case 2 says \( T(n) = \Theta \log n \).
Master Method: Example 7

\[
T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}
\]

Theorem (The Master Theorem)

1. if \( f(n) \) is \( O(n^{\log_b a-\varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a+\varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[
T(n) = 2 \cdot T(n/2) + \log n \quad \text{heap construction}
\]

Solution: \( \log_b a = 0 \), thus case 1 says \( T(n) = \Theta(n) \).