COMP251: Review

Jérôme Waldispühl
School of Computer Science
McGill University
Overview of the exam

• 11 questions.
• 200 points + 30 bonus (last question)
• 20 point True or False (Warning penalty for false answers!)
• 20 point for *multiple* choices answers.
• 28 points for short answers (no justification)
• 97 points questions/applications
• 35 points + 30 bonus problems
• Unless specified, all answers must be justified.
• Partial answers will receive credits.
• The clarity and presentation of your answers is an integral part of the grading. **Be neat!**
• Books and electronic devices are not allowed.
• 6 crib sheets (12 pages).
Material for the exam

• Everything from lecture 1 to lecture 21 except Fast Fourier Transform.

• Lecture on data compression & pattern matching are not included in the exam.

• Strongly connected components will not be included in the final exam either.
Midterm review
True or False

(a) (2 points) The worst case search time with an hash table using open addressing is $O(1)$.
   A. True  B. False

(b) (2 points) At least half of the nodes on a path from the root to a leaf of a red-black trees are black.
   A. True  B. False

(c) (2 points) We run the depth-first search algorithm (DFS) on a graph $G$ and found one back edge. Thus, $G$ has a cycle.
   A. True  B. False

(d) (2 points) Every directed acyclic graph has exactly one topological ordering.
   A. True  B. False

(e) (2 points) A sub-graph that respects the cut has no light edge.
   A. True  B. False

(f) (2 points) Dijkstra’s algorithm may not terminate if the graph contains a negative-weight cycle.
   A. True  B. False

(g) (2 points) A bipartite graph has no cycle.
   A. True  B. False

(h) (2 points) In a flow network such that the source vertex $s$ has no incoming edge and the sink $t$ no outgoing edges, the sum of the flow out of $s$ is equal to the flow in $t$.
   A. True  B. False
Multiple Choices

(a) (4 points) What is the maximum number of keys that you can store in a binary heap of height $h$?
A. $2^h$  B. $2^{h-1}$  C. $2^h - 1$

(b) (4 points) We relax the following edge during the execution of the Dijkstra. The shortest path estimate before relaxation of the edge is stored inside the vertices. What will be the new value of the shortest path estimate of the rightmost vertex after relaxation of this edge?
A. 12  B. 10  C. 5  D. 14
(c) (4 points) We are in the middle of the execution of the Kruskal algorithm for computing a minimal spanning tree (MST) of the graph below. The bold edges are the edges already selected to be in the MST. The edge \((d, e)\) has just been added. What will be the next one to be selected?

A. \((c, d)\)  
B. \((a, b)\)  
C. \((d, f)\)  
D. \((f, h)\)
(e) (4 points) What is the longest series of activities (i.e. number of activities) returned by the greedy algorithm for solving the scheduling problem (i.e. finding the maximal number of compatible activities) that we have seen in class?

A. \((a_7, a_5, a_2, a_6, a_3)\)  
B. \((a_1, a_5, a_2, a_6, a_9)\)  
C. \((a_4, a_5, a_8, a_3, a_9)\)  
D. \((a_4, a_5, a_2, a_6, a_3)\)
(f) (4 points) We consider the flow network below, where all edges have a capacity equal to 1. Let $a$ be the source vertex and $h$ the sink. Which one(s) of the following cuts (i.e. partition of the vertices) is/are minimal cut(s)?

A. $\{a, b, c, d, e\}, \{f, g, h\}$
B. $\{a, b, c, e\}, \{d, f, g, h\}$
C. $\{a\}, \{b, c, d, e, f, g, h\}$
D. $\{a, b, c, e, d, g\}, \{f, h\}$
E. $\{a, b, c, d, e, f\}, \{g, h\}$

![Flow network diagram]
AVL trees

Insert 12 and restore AVL properties.

Left rotation at 10
AVL trees

Insert 12 and restore AVL properties.

Right rotation at 24
Hash tables

Hash function \( h(x) = x \mod 7 \). Insert 20, 9, 7, 3, 27, 15, and 19.

<table>
<thead>
<tr>
<th>Direct Addressing</th>
<th>Open Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>
Flow networks

Residual graph?
Flow networks

Augmenting path?

Value of augmenting path = minimum capacity along that path.
Flow networks

What is the flow network once we added the augmenting path?

Is the flow maximal? Yes. Consider the cut \{s,u,v,x,y\},\{t\} which has a capacity of 7. The value of the flow is already 7 and therefore the flow cannot be augmented.
Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex $s$ may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra’s algorithm correctly finds shortest paths from $s$ in this graph.

From the proof of correctness (slides 43-47 of Lecture 10), we want to show that the maintenance property is satisfied. In other words, $d[u] = \delta(s,u)$ when $u$ is added to $S$ in each iteration.”

Other part of the proof (Initialization, Loop invariant and Termination are still satisfied.)
Show that \( d[u] = \delta(s,u) \) when \( u \) is added to \( S \) in each iteration.

We will show a contradiction. We will follow here the same proof and same notations we used in class!

Suppose there exists \( u \) such that \( d[u] \neq \delta(s,u) \).

Let \( u \) be the first vertex for which \( d[u] \neq \delta(s,u) \) when \( u \) is added to \( S \).

- \( u \neq s \), since \( d[s] = \delta(s,s) = 0 \).
- Therefore, \( s \in S \), so \( S \neq \emptyset \).
- There must be some path \( s \leadsto u \). Otherwise \( d[u] = \delta(s,u) = \infty \) by no-path property.
- So, there is a path \( s \leadsto u \). Thus, there is a shortest path \( p \in S \leadsto u \).
Show that $d[u] = \delta(s,u)$ when $u$ is added to $S$ in each iteration.”

Just before $u$ is added to $S$, path $p$ connects a vertex in $S$ (i.e., $s$) to a vertex in $V - S$ (i.e., $u$). Let $y$ be first vertex along $p$ that is in $V - S$, and let $x \in S$ be $y$'s predecessor (Eventually, you could have $x=s$ or $y=u$).

Decompose $p$ into $s \sim x \rightarrow y \sim u$. 
Single-source shortest path

``Show that $d[u] = \delta(s,u)$ when $u$ is added to $S$ in each iteration.''

Now can get a contradiction to $d[u] \neq \delta(s, u)$:

- $y$ is on shortest path $s \leadsto u \Rightarrow d[y] = \delta(s,y)$ -- by convergence property (See Lecture 10).
- Thus, $d[y] = \delta(s,y)$.
- $\delta(s,y) \leq \delta(s,u)$ because all edge weights after $y$ are nonnegative (Otherwise, we would have a shortest path $s \leadsto y \leadsto s \leadsto u$, but $s \leadsto y \leadsto s$ is a nonnegative cycle thus contradiction).
- $\delta(s,u) \leq d[u]$ by upper-bound property $\Rightarrow d[y] \leq d[u]$.
- Since $y$ and $u$ were in $Q$ when we chose $u$, then: $d[u] \leq d[y]$
- Thus $d[u] = d[y]$, and $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.
- This contradicts assumption that $d[u] \neq \delta(s,u)$. ■
Dynamic programming
Knapsack problem

- Given $n$ objects and a "knapsack."
- Item $i$ weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of $W$.
- Goal: fill knapsack so as to maximize total value.

<table>
<thead>
<tr>
<th></th>
<th>$i$</th>
<th>$v_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Knapsack instance (weight limit $W = 11$)

Ex. $\{1, 2, 5\}$ has value 35.
Ex. $\{3, 4\}$ has value 40.
Ex. $\{3, 5\}$ has value 46 (but exceeds weight limit).

Greedy by value. Repeatedly add item with maximum $v_i$.
Greedy by weight. Repeatedly add item with minimum $w_i$.
Greedy by ratio. Repeatedly add item with maximum ratio $v_i / w_i$.

Observation. None of greedy algorithms is optimal.
Def. \(OPT(i) = \text{max profit subset of items } 1, \ldots, i\).

Case 1. \(OPT\) does not select item \(i\).
   - \(OPT\) selects best of \(\{1, 2, \ldots, i - 1\}\).

Case 2. \(OPT\) selects item \(i\).
   - Selecting item \(i\) does not immediately imply that we will have to reject other items.
   - Without knowing what other items were selected before \(i\), we don't even know if we have enough room for \(i\).

Conclusion. Need more subproblems!
Def. \( OPT(i, w) = \text{max profit subset of items 1, \ldots, } i \text{ with weight limit } w. \)

Case 1. \( OPT \) does not select item \( i \).

- \( OPT \) selects best of \( \{1, 2, \ldots, i-1\} \) using weight limit \( w \).

Case 2. \( OPT \) selects item \( i \).

- New weight limit = \( w - w_i \).
- \( OPT \) selects best of \( \{1, 2, \ldots, i-1\} \) using this new weight limit.

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise}
\end{cases}
\]
Dynamic programming algorithm

**Knapsack** \((n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)\)

**FOR** \(w = 0\) **TO** \(W\)

\[ M[0, w] \leftarrow 0. \]

**FOR** \(i = 1\) **TO** \(n\)

**FOR** \(w = 1\) **TO** \(W\)

**IF** \((w_i > w)\)

\[ M[i, w] \leftarrow M[i-1, w]. \]

**ELSE**

\[ M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}. \]

**RETURN** \(M[n, W].\)
Example

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>(v_i)</th>
<th>(w_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>weight limit (w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>{}</td>
</tr>
<tr>
<td>{1}</td>
</tr>
<tr>
<td>{1, 2}</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
</tr>
</tbody>
</table>

\(OPT(i, w) = \max\) profit subset of items \(1, \ldots, i\) with weight limit \(w\).
Theorem. There exists an algorithm to solve the knapsack problem with \( n \) items and maximum weight \( W \) in \( \Theta(nW) \) time and \( \Theta(nW) \) space.

Pf.
• Takes \( O(1) \) time per table entry.
• There are \( \Theta(nW) \) table entries.
• After computing optimal values, can trace back to find solution:
  take item \( i \) in \( OPT(i, w) \) iff \( M[i, w] < M[i - 1, w] \).

Remarks.
• Not polynomial in input size! "pseudo-polynomial"
• Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
• There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]
Amortized Analysis
Dynamic tables

**Scenario**
- Have a table - maybe a hash table.
- Don’t know in advance how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
- When it gets sufficiently small, *might* want to reallocate with a smaller size.

**Goals**
1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

**Load factor** $\alpha = (\# \text{ items stored}) \div (\text{allocated size})$

Never allow $\alpha > 1$; Keep $\alpha >$ a constant fraction $\implies$ Goal 2.
Table expansion

Consider only insertion.

• When the table becomes full, double its size and reinsert all existing items.
• Guarantees that $\alpha \geq \frac{1}{2}$.
• Each time we insert an item into the table, it is an elementary insertion.

TABLE-INSERT($T, x$)

if $size[T] = 0$
  then allocate $table[T]$ with 1 slot
  $size[T] \leftarrow 1$

if $num[T] = size[T]$ then
  allocate new-table with $2 \cdot size[T]$ slots
  insert all items in $table[T]$ into new-table
  free $table[T]$
  $table[T] \leftarrow$ new-table
  $size[T] \leftarrow 2 \cdot size[T]$
insert $x$ into $table[T]$

$num[T] \leftarrow num[T] + 1$  \quad \text{(Initially, } num[T] = size[T] = 0)
Aggregate analysis

- Charge 1 per elementary insertion.
- Count only elementary insertions (other costs = constant).

$c_i =$ actual cost of $i^{th}$ operation

- If not full, $c_i = 1$.
- If full, have $i-1$ items in the table at the start of the $i^{th}$ operation. Have to copy all $i-1$ existing items, then insert $i^{th}$ item $\Rightarrow c_i = i$.

$n$ operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for $n$ operations

$$c_i = \begin{cases} 
  i & \text{if } i-1 \text{ is power of 2} \\
  1 & \text{Otherwise}
\end{cases}$$

Total cost $= \sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j = n + \frac{2^{\lfloor \log n \rfloor + 1} - 1}{2 - 1} < n + 2n = 3n$

Amortized cost per operation $= 3$. 
Accounting method

Charge $3 per insertion of x.
• $1 pays for x’s insertion.
• $1 pays for x to be moved in the future.
• $1 pays for some other item to be moved.

Suppose we’ve just expanded, $size=m$ before next expansion, $size=2m$ after next expansion.
• Assume that the expansion used up all the credit, so that there’s no credit stored after the expansion.
• Will expand again after another $m$ insertions.
• Each insertion will put $1 on one of the $m$ items that were in the table just after expansion and will put $1 on the item inserted.
• Have $2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion...