Eksamen emne 5 – Algoritmiske metoder

Oppgave 1.1

**Insertion sort** er en enkel sorterings algoritme som lager det sorterte arrayet et element av gangen. Den starter med et tomt array og hvert element i input dataen sammenlignes med de allerede sorterte elementene og plasserer ny input på rett plass.

Vi starter med et array av tallene 4, 2, 3, 1. Målet er å sortere dem i stigende rekkefølge. Først sammenligner vi de to første tallene, 4 og 2. Siden 4 er større enn 2 og vi ønsker en stigende rekkefølge, bytter vi plass på disse. Arrayet blir da 2, 4, 3, 1.

Neste sammenligner vi det andre og tredje tallet, som nå er 4 og 3. Igjen er 4 større enn 3, så vi bytter plass på disse. Listen er nå 2, 3, 4, 1.

Til slutt sammenligner vi 4 og 1. Siden 4 er større enn 1, bytter disse plass. Arrayet er nå 2, 3, 1, 4. Nå er ikke arrayet sortert i stigende rekkefølge igjen, så da sammenlignes 1 med 3 og bytter plass med 3 og det samme med 2 og bytter plass med den. Til slutt står vi igjen med listen 1, 2, 3, 4.

**Heap Sort** er en sorterings algoritme som også baserer seg på sammenligning. Den bruker en binær heap-data struktur for å bygge en max-heap eller min-heap. Så algoritmen vil først lage et heap fra input arrayet og deretter gjentatte ganger fjerne det største(max) eller minste(min) elementet, og til slutt ende med et sortert array. Et bilde som inneholder håndskrift, line, tekst, diagram

Automatisk generert beskrivelseEksempel Max-Heap

Henviser til tegning over. Som eksempel kan vi bruke arrayet 2, 8, 5, 4, 1. Først lager man et «tre», med 2 i toppen som har grener til 8 og 5, og 8 har grener til 4 og 1. Deretter bytter vi plass på 8 og 2, og 2 og 4 (figur 2 og 3).

Deretter fjerner vi 8 og plasserer den i det tomme sorterte arrayet. Så tar vi 1 fra grenen til 2 og plasserer i toppen og kjører samme prosedyre på nytt til vi ender med det høyeste tallet i toppen som nå er 5 (figur 4, 5 og 6). Deretter blir 5 plassert foran 8 i arrayet.

Nå havner 2 i toppen, så vi bytter plass på 4 og 2 og plasserer 4 foran 5 i arrayet. Nå står vi igjen med 2 og 1, der to er over 1 og 2 blir derfor plassert foran 4 i arrayet (figur 7, 8, 9 og 10). Til slutt står vi igjen med bare tallet 1, og det blir da plassert helt foran i arrayet.

**Quicksort** er en effektiv sorteringsalgoritme som velger et element å sammenligne (pivot) seg med, og sorterer ved å jobbe seg rundt det valgte elementet for å få det til å ende opp på riktig plass.

Her bruker vi array 1, 7, 2, 4 som eksempel. Vi kan bruke 4 som pivot, så kan vi sammenligne 1 med 4, og siden en er mindre blir 1 stående på plassen sin. Deretter kan vi ta 7, siden 7 er høyere enn 4 hopper vi videre til 2. Siden 2 er mindre enn 4 plasserer vi 2 på plassen sin, som vil si at vi bytter plass på 2 og 7.

Arrayet er nå 1, 2, 7, 4. Nå bytter 7 og 4 plass slik at pivoten står på korrekt plass. Det sorterte arrayet er 1, 2, 4, 7.

Oppgave 1.3

Valget av pivot påvirker i stor grad effektiviteten av QuickSort algoritmen. Dersom du for eksempel alltid velger det første eller siste tallet i arrayet/ listen, kan det føre til ujevne partisjoner. Det vil si at en partisjon er mye mindre eller større enn den andre. Da vil algoritmen bruke mer tid på å sortere den store, noe som fører til at det totalt sett tar lenger tid. I verste fall kan tidsløpet gå til O(n^2), noe som gjør at Quicksort stiller på linje med Bubble Sort, Selection sort eller Insertion Sort, som alle har worst-case og average-case complexity på O(n^2) når det kommer til effektivitet. Det vil si at et dårlig valg av pivot kan føre til at man undergraver alle styrkene til algoritmen, som er avhengig av en jevn fordeling i partisjonene. Average- case i Quicksort er O(n log n), n representerer antall elementer i arrayet. Noe som også vil si at tidsforløpet vil være lineært med antall elementer i inputen.

Den optimale pivoten vil være medianen eller så nære medianen som mulig, derfor kan man minimere risikoen av velge en dårlig pivot ved å enten ta et helt tilfeldig tall eller plukke ut tre tilfeldige tall, og ta medianen av de tre tallene. På den måten er det større sannsynlighet for å treffe nærmere den reelle medianen.

Til slutt kan man si at et dårlig valg av pivot i Quicksort algoritmen kan føre til flere sammenligninger og ujevne partisjoner, som igjen fører til worst-case complexity på O(n^2). Derfor blir valget av pivot avgjørende for effektiviteten til algoritmen.

Oppgave 2.1

X -> A -> B -> N -> T -> H -> G -> Y

Oppgave 2.3

Både Dijkstras Algoritme og A\* pathfinding er algoritmer som har som mål å finne den korteste veien mellom noder i en vektet graf. Forskjellen er at Dijkstras Algoritme er innom alle noder i grafen og kalkulerer fra kildenoden og beveger seg bortover for å finne den garantert korteste veien. Ulempen med den algoritmen er at den er mindre effektiv, derfor passer denne algoritmen best når man er avhengig av å kalkulere hele grafen for å være helt sikker på å finne den korteste veien. For eksempel vil en graf med 100 noder gå igjennom ca 10 000 kalkulasjoner, noe som vil være kreve mye tid. A\* bruker mye av de samme prinsippene som Dijkstras algoritme, men i stede for å jobbe seg gjennom og kartlegge hele grafen vil den hele tiden bruke heuristikk, som vil si at algoritmen estimerer hva det vil koste å nå målet fra en gitt node. Dijkstras Algoritme blir da en mer komplett algoritme som alltid vil komme frem til det riktige svaret, mens A\* kan gi mer unøyaktige resultater. På tross av det er ofte A\* foretrukket i de situasjonene der man har spesifiserte start- og slutt-punkter, da denne algoritmen kan være mer effektiv.

https://courses.lumenlearning.com/waymakermath4libarts/chapter/shortest-path/

https://www.baeldung.com/cs/dijkstra-vs-a-pathfinding